Detecting Prolog Programming Techniques Using Abstract Interpretation

Andrew Bowles

Ph.D.
University of Edinburgh
1991

I declare that this thesis has been composed by myself and that the work it describes is my own.

Abstract

There have been a number of attempts at developing intelligent tutoring systems (ITSs) for teaching students various programming languages. An important component of such an ITS is a debugger capable of recognizing errors in the code the student writes and possibly suggesting ways of correcting such errors. The debugging process involves a wealth of knowledge about the programming language, the student and the individual problem at hand, and an automated debugging component makes use of a number of tools which apply this knowledge. Successive ITSs have incorporated a wider range of knowledge and more powerful tools.

The research described in this thesis should be seen as carrying on with this succession. Specifically, we attempt to enhance an existing Prolog ITS (PITS) debugger called APROPOS2 developed by Looi. The enhancements take the form of a richer language with which to describe Prolog code and more powerful tools with which constructs in this language may be detected in Prolog code.

The richer language is based on the notion of programming techniques—common patterns in code which capture in some sense an expert's understanding of Prolog. The tools are based on Prolog abstract interpretation—a program analysis method for inferring dynamic properties of code. Our research makes contributions to both these areas.

We develop a language for describing classes of Prolog programming techniques that manipulate data-structures. We define classes in this language for common Prolog techniques such as accumulator pairs and difference structures.

We use abstract interpretation to infer the dynamic features with which techniques are described. We develop a general framework for abstract interpretation which is described in Prolog, so leading directly to an implementation. We develop two abstract domains—one which infers general data flow information about the code and one which infers particularly detailed type information—and describe the implementation of the former.

Acknowledgements

I would like to give my sincerest thanks to my supervisor Chris Mellish for his help and advice over the years of my research.

I would also like to thank the following for their help and comments on drafts of my work: Diana Bental, Paul Brna, Tim Duncan, Helen Pain, Brian Ross, Rob Scott, Jussi Stader and Paul Wilk.

"Many years may be spent in the search after knowledge,
but if the right path is hit at first,
it may be attained in a little time"

Contents

Chapter 1

Introduction

There have been a number of attempts at developing intelligent tutoring systems (ITSs) for teaching students various programming languages [35, 56, 46]. An important component of such an ITS is a debugger capable of recognizing errors in the code the student writes and possibly suggesting ways of correcting such errors. The debugging process involves a wealth of knowledge about the programming language, the student and the individual problem at hand, and an automated debugging component makes use of a number of tools which apply this knowledge. Successive ITSs have incorporated a wider range of knowledge and more powerful tools.

The background of the research described in this thesis should be seen as carrying on with this succession. Specifically, we investigate enhancements of an existing Prolog ITS (PITS) debugger called APROPOS2 developed by Looi [46]. The enhancements take the form of a richer language with which to describe Prolog code and more powerful tools with which constructs of this language may be detected in Prolog code.

The richer language is based on the notion of programming techniques—common patterns in code which capture in some sense an expert's understanding of Prolog. The tools are based on Prolog abstract interpretation—a program analysis method for inferring dynamic properties of code. Our research makes contributions to both these areas.

We now describe APROPOS2 and show the limitations it has which our research attempts to overcome.

1.1 An Overview of APROPOS2

APROPOS2 implements an algorithm-based approach to inferring what a novice's program is intended to do and how it is intended to work, and identifying errors in these intentions and their realizations. "Algorithm-based" means that APROPOS2 attempts to infer the algorithm the student intended to use to solve the task. APROPOS2 ex-

amines a program (which is assumed to be syntactically correct) in two stages.

The first stage performs multiple program analyses on the program but makes no use of knowledge about what the program is intended to do. Some of these analyses can detect certain program anomalies, but the information derived is mostly used in the second stage of APROPOS2's execution. The program analyses used include the following:

Mode Analysis Mode analysis infers the instantiation pattern of the calls to the predicates in the program [49, 22]. This analysis can detect bugs dependent on the instantiation state of variables—for example, uninstantiated variables in arithmetic built ins.

Dataflow Analysis Dataflow analysis reasons about the control flow of the program and can detect unreachable code and non-terminating loops caused by missing or incorrect base cases.

Type Analysis Type analysis attempts to characterize the form of the arguments to each call and exit in a program [60, 54].

Misspelling Analysis Spelling mistakes in Prolog do not cause compile time errors but result in programs with unintended meanings. Symptoms of such mistakes are references to undefined predicates and singleton variable warnings. APROPOS2 uses a knowledge of misspellings to try to correct these bugs.

Two further analyses, which make use of some of the information derived by the above analyses, are also performed. The first attempts various program transformations in an effort to simplify the student's code. For example, code of the form 'append([H], T, X)' can be replaced by the unification 'X=[H|T]'. The second tries to detect certain common code patterns, known as programming techniques. APROPOS2 defines a number of simple programming techniques. For example, it recognizes the use of the same variable in the same argument position of a predicate in a recursive clause to pass back an accumulated result.

The output of this first stage is (in addition to the program anomalies listed above) an internal representation of the student's program known as a Prolog frame, or P-frame. This is used as an input to the second stage.

The second stage uses task-specific knowledge (also stored as a P-frame) to analyse the program, detect bugs and correct them. Task-specific knowledge is represented at three levels of abstraction:

Algorithm Level A programming problem will have a number of solutions based on different algorithms. Bugs at this level imply a misunderstanding of the task.

APROPOS2 contains descriptions of correct algorithms for each task, as well as descriptions of common incorrect or inefficient algorithms.

Predicate Definition Level The bugs at this level involve missing, extra or incorrect predicate definitions given a particular algorithm.

Implementation Level The bugs at this level involve coding errors such as missing or extra clauses, missing or extra subgoals and clauses and subgoals in the wrong order.

These three levels relate to APROPOS2's debugging approach; that is, it first decides which algorithm has been used, then how this has been decomposed into predicates and then how each predicate is implemented. The following is taken from Looi's thesis [46] (pages 62-3):

> APROPOS2 attempts to localize the above kinds of bugs by a process of heuristic code-matching and code-critique, and then to provide a description of the bugs. The bug description is more than a listing of where the student's incorrect Prolog program differs syntactically from some correct program. As code-matching is done top-down from task statement to algorithm to predicate definition decomposition and then to implementation of each predicate definition, APROPOS2 postulates what the student is intending to do for each line of code he has written. ... In this way, we have attempted to relate the commentaries to the program context which helps to relate the bugs to possible misconceptions.

The heuristic search matches features in the P-frames for the student's code with features in reference P-frames for each implementation of an algorithm. The match is essentially syntactic, but APROPOS2 can perform some dynamic checks to prove the equivalence of predicate calls. A score function is used to evaluate each match against each P-frame. The algorithm described by the P-frame with the highest score is treated as being the algorithm intended by the student, and any mismatches are treated as bugs.

1.1.1 Case Study: reverse/2

We now summarize the case study of APROPOS2's analysis given by Looi. The following is a simplified description of APROPOS2's analysis. The task and student's solution are given in Figure 1.1.

The first, multiple analysis stage of APROPOS2's critique infers the following:

- The mode of the call to append/3 predicate in the definition of reverse/2 is append(+,+,-).

Task: **LIST-REVERSAL** Write a Prolog program reverse/2 which takes a list as input and reverses the elements of the list. A goal "reverse(X,Y)" should succeed with Y instantiated to the reverse of the list X.

```
append([], L, L).
append([H], L, [H|L]).
append([H|T], L, Ans) :-
   append(T, L, X),
   append([H], X, Ans).
reverse([], []).
reverse([H], [H]).
reverse([H|T], L) :-
   reverse(T, M),
   append(M, H, L).
```

Figure 1.1: The reverse/2 task and a student's buggy solution

---

- All arguments to append/3 should be lists.

Also the following program transformation is detected:

```
Your clause:
append([H|T], L, Ans) :-
   append(T, L, X),
   append([H], X, Ans).
can be rewritten as:
append([H|T], L, [H|X]) :-
   append(T, L, X).
```

APROPOS2 has three P-frame representations for algorithms for list-reversal. Figure 1.2 shows the key features of the P-frame for naive reverse.

APROPOS2 then uses its task knowledge to recognize the algorithm used in the program. APROPOS2 selects the naive reverse algorithm as being the most similar to the student's solution. The student's reverse/2 predicate is mapped onto the reference representation of reverse/2. This marks a number of clauses as being redundant since there is no corresponding clause descriptor in the reference P-frame. There is a mismatch between the 'append(M,H,L)' subgoal in the code and the reference 'append(M,[H],L)'. APROPOS2 will try to show their equivalence by running test cases and comparing results. This dynamic analysis fails to find an equivalence, so APROPOS2 concludes that an incorrect subgoal bug has been made.

APROPOS2's output for this analysis is given in Figure 1.3. Note that the text output is simple 'canned' text. In this case, the bug in the program is handled by a special buggy clause description for this bug.

TASK NAME: reverse/2
ALGORITHM NAME: naive-reverse
LIKELY PREDICATE NAMES: reverse, rev, rv
INVOCATION TYPE OF PREDICATE: both arguments are lists
INVOCATION MODE OF PREDICATE: reverse(+,-)
PROGRAMMING TECHNIQUES: naive-recursion
RECURSION ARGUMENT: 2
CLAUSE 1:
TYPE: base
HEAD GOAL: reverse([],[])
PREFIX SUBGOALS: nil
RECURSIVE SUBGOALS: nil
SUFFIX SUBGOALS: nil
COMMENTARY: "This base case says that the reverse of the empty list is the empty list."
CLAUSE 2:
TYPE: recursive
HEAD GOAL: reverse([H|T],Res)
PREFIX SUBGOALS: nil
RECURSIVE SUBGOALS: reverse(T,Sofar)
SUFFIX SUBGOALS: append(Sofar,[H],Res)
COMMENTARY: "This recursive case says that the reverse of a non-empty list can be found by reversing the tail of the list and then appending a list consisting only of the first element of the original to the end of the reverse of the tail."
... *more clause representations* ...
CLAUSE ORDERING: no constraint
TEST CASES:
[reverse([],Res), Res=[]]
[reverse([a,b],Res), Res=[b,a]]
[reverse([a,b,c,d,e],Res), Res=[e,d,c,b,a]]
[append([],[a],Res), Res=[a]]
[append([b],[a],Res), Res=[b,a]]
[append([e,d,c,b],[a],Res), Res=[e,d,c,b,a]]

Figure 1.2: P-frame representation for naive reverse.

1.2 Improving APROPOS2

In its limited context APROPOS2 is successful and certainly demonstrates the validity of the algorithmic-based approach. However, Looi acknowledges that some of his analysis techniques are largely *ad hoc*. He suggests the use of abstract interpretation to place the analysis methods on a more principled foundation. This forms part of the motivation for the work described in this thesis.

We also suggest that APROPOS2's P-frame language used for describing programs is somewhat limited. For example, it is not clear whether certain common Prolog programming techniques such as accumulator pairs and difference structures may be described accurately[1]. The problem of developing a more expressive and principled

[1]The P-frame language is not described precisely enough to judge this. However, Looi does not describe difference list implementations of algorithms which often involve difference lists.

```
Your program seems to be incorrect.
Now, this is APROPOS2's critique on your program:

Clause 1 of reverse/2 seems fine.
This base case says that the reverse of the empty list is the empty list.

Clause 2 of reverse/2
   reverse([H],[H]).
seems redundant.

In Clause 3 of reverse/2
   append(M,H,L) is incorrect.
Replace it by append(M,[H],L).
This recursive case says that the reverse of a non-empty list can be found by
reversing the tail of the list and then appending a list consisting only of
the first element of the original to the end of the reverse of the tail. For
the append goal to work, it needs arguments which are of type list. In this
instance, H is an element which is to be tagged to the end of another list M.
So we need to write append(M,[H],L).

Clause 1 of append/3 seems fine.
This base case says that the result of appending an empty list to a list L is
the list L.

Clause 2 of append/3
   append([H],L,[H|L]).
seems redundant.

Clause 3 of append/3 seems fine.
This recursive clause says that the result of appending a list [H|T] to a list
X is obtained by appending [H] to the result of appending the list T to X.
```

Figure 1.3: APROPOS2's analysis of a student's naive reverse/2 program.

language for describing (parts of) programs is the second motivation of our work. The immediate aim is to develop a language which can accurately represent common Prolog constructs for manipulating data-structures. This class of techniques is well suited to APROPOS2 because many of the example tasks it currently has knowledge about involve simple 'list processing' where data manipulation is most obvious. We expect that such a language will have other benefits. For example, it is currently not obvious how well larger programs may be represented as P-frames; a more expressive language will certainly help with this.

Another problem we might expect an improved P-frame language to address is seen in the difficulty APROPOS2 has in handling the following implementation of reverse/2:

```
reverse(A, B) :-
   rev(A, [B]).
rev([], [B|B]).
rev([Head|Tail], [B|Temp]) :-
   rev(Tail, [B,Head|Temp]).
```

This implements the usual quick reverse algorithm, but uses the list constructor ./2 to form a pair of the accumulator input and result, leading to particularly obscure code. APROPOS2 fails to match the accumulator and produces a series of 'incorrect clause' errors even though its dynamic analysis shows that the student's program produces correct results. Looi concludes ([46], page 136):

> A more canonical representation of program dataflow may be able to match different implementations of the same dataflow ...

We place our more expressive language in the context of work on Prolog programming techniques. Looi rightly rejects a purely techniques based approach to bug diagnosis ([46], page 23):

> The automatic detection of the use of techniques in a student's program is a difficult task because the student may have used a technique incorrectly or used an inappropriate technique. An automated debugger needs some expectation of what the student is intending to do.

We claim that a language of programming techniques is precisely what is needed to enhance the current P-frame language. The problem of detecting techniques is an important one, but is separate from the problem of providing an appropriate language for describing programs. We now introduce our notion of programming techniques.

1.3 Prolog Programming Techniques

Prolog's execution is based on three mechanisms: unification, procedure call and search based on backtracking. Out of these mechanisms it is possible to express a wide range of conventional program constructs—for example, case based choice and recursion—as well as some more unconventional ones, such as failure driven loops and difference structures. An expert Prolog programmer will possess a good grasp of how to combine the basic mechanisms into some pattern to implement whatever construct is needed. Indeed, a common complaint often heard from Prolog experts is the monotony of having to write down 'by hand' these constructs in low level terms every time one is used. On the other hand, students encounter a great deal of difficulty in learning the correct pattern for each construct [2, 75]. These observations have given rise to the idea of *programming techniques* as a way of expressing this knowledge of Prolog. The case for the importance of programming techniques is given by Brna *et al* [6] where questions are discussed such as: what can techniques be used for, how is the set arrived at and how is it structured, and how are techniques to be represented?

We explain the notion of techniques by considering the following decomposition of the problem solving process. A problem is initially stated as a specification, and

the first task a programmer has is to develop an algorithm to meet this specification. This is, of course, a difficult task but one which mainly involves reasoning about the problem at hand, rather than the programming language the eventual solution will be written in. An algorithm is a description of how to meet some specification which is more or less problem specific but implementation independent. The notion of plan [67] is closely related to that of algorithm as used here.

The next task a programmer has is to write a program in some language which implements the algorithm. This involves using general knowledge about programming—for example, the complexity of accessing different data-structures—as well as specific knowledge about the programming language being used, such as ways of achieving certain effects in the language. Techniques form part of this language specific knowledge. A technique is some common programming practice which has some particular behaviour but which is independent of the algorithm being implemented and, therefore, independent of the problem at hand.

As an example, consider the (standard) problem of mapping (that is, flattening) a nested list into a flat list. Part of an algorithm for this involves recursively flattening the head of the list and the tail of the list and concatenating the results of the each together. The concatenation may be implemented in Prolog by an explicit call to a predicate such as the standard append/3 predicate, or by using the difference list technique which takes advantage of Prolog's logical variables to implement concatenation in an efficient way [69]. The difference list solution is:

```
flatten([H|T], L-L0) :-
    flatten(H, L-L1),
    flatten(T, L1-L0).
flatten([], L-L).
flatten(X, [X|L]-L) :-
    X\=[], X\=[_|_].
```

where the two parts of the difference list are bundled together into the second argument of flatten/2. This is an example of a technique for building data-structures. Techniques can also capture common control flow constructs, such as generate-and-test.

A related notion is that of program *clichés* [76, 5] which includes both techniques and algorithms. Clichés arise from the more pragmatic viewpoint that an experienced programmer has a large library (often only a mental library) built up over a period of time containing many convenient code generalizations. The important distinction between clichés and techniques is that whereas the set of clichés is large, the set of techniques should be relatively small. Brna [6] expects to discover of the order of one hundred common techniques. It seems likely that most Prolog programs may be generated by combination and specialization of a handful of this set of techniques, although it is difficult to judge this objectively.

Rather than viewing techniques as aids to writing programs, we can conversely view them as ways of decomposing and so *explaining* programs. This viewpoint makes them appropriate for our task of enhancing P-frames. We envisage an enhanced P-frame language which includes slots specifying certain techniques which should appear in particular clauses and predicates.

There are two requirements for allowing APROPOS2 to make use of techniques:

- We need a language with which to describe techniques. Typically techniques are described (as we have done above) by use of an example. This is not sufficient because it describes very specific instances of techniques. In APROPOS2, where algorithms are being described, many details are not important. For example, in the flatten/2 example above the name of the -/2 constant used to bundle the two halves of the difference list together is not important. The language should be able to describe techniques at a greater or lesser degree of abstraction so that algorithms may be described with an appropriate level of detail.

- We need to be able to detect instances of techniques in student's code. In APROPOS2, it must be possible to infer likely technique instances in code so that a P-frame representing the student's code may be constructed and then matched against algorithm P-frames. Note that because of the lack of syntactic features in Prolog a techniques detector based purely on a syntax matcher may well be easily misled, particularly with techniques that are slightly buggy.

In Chapter 3 of this thesis we develop a language for describing a set of common Prolog programming techniques for manipulating data-structures which we claim is suitable for enhancing APROPOS2's P-frame language. Such techniques are characterized in terms of the dynamic effects of unifications, along with other static features such as syntax. Different classes of technique are defined by particular patterns of these effects. The effects of unifications are specified in terms of two relationships between the variables in clauses: inclusion and sharing.

We define a hierarchy of technique classes which demonstrates that the common Prolog techniques of difference structures and accumulator pairs can be seen as different instances of a more fundamental class of techniques. The differences between these two technique classes are dependent on non-declarative aspects of Prolog, in particular the input and output mode of arguments and the order of subgoals in clause bodies. In practice, most techniques definitions are dependent on non-declarative features.

Techniques may be detected in code using a simple syntactic match between the code and the technique description provided that the code has been annotated with the dynamic features—that is, inclusion and sharing relations. To infer these features we use abstract interpretation.

1.4 Abstract Interpretation

There are many situations where it is desirable to know some property of the states of execution a program may reach that is not immediately obvious from the program text. For example, if it is known that a predicate is always called with its first argument ground then a compiler can save several instructions in the code it produces [50]. An obvious way to work out these properties would be to actually run the program and make a note of the states that it gets into. While this is plausible when it is possible to perform the analysis 'by hand', there are some problems in using this approach to construct automatic program analysers:

- To be able to make a strong statement such as "this argument is always ground" the inputs to the program would need to be carefully chosen. Generally, the set of example inputs would need to be infinite to provide a guarantee of such statements.
- Because they can involve infinite behaviours, programs do not always terminate.

A solution to both of these problems is to step back from the details of how a program is executing and consider only the execution in terms of the properties of interest; that is, consider only an abstraction of a program's execution. This is *abstract interpretation.* In doing this abstraction the 'size' of the domain over which the interpreter is executing can be very much reduced, avoiding the above problems. If the usual interpreter executes over the *concrete* domain of Prolog terms, then an abstract interpreter executes over this reduced *abstract* domain of properties of Prolog terms.

For example, if we are interested only in the groundness of terms we could ignore the particular constants used to make up those terms and simply record the groundness of each variable in the substitution built by the abstract interpreter. To illustrate this, consider the execution of the following program:

```
p(f(A), B) :- q(A), C=g(A), r(C), D=g(B), s(D).
?- p(f(a), X).
```

After the successful head match, the variable A is bound to the atom a/0. Being an atom, it is ground. This means that the call to the predicate q/1 is completely ground, as is the call to r/1 because the variable C is constructed out of ground components. The call to s/1, on the other hand, involves the variable B which is still uninstantiated and so this call is not ground.

With an abstract interpreter we can execute this program with regard only to this groundness information. During execution, instead of constructing the usual Prolog terms, two special tokens are used; g representing all ground terms and a representing all Prolog terms. Thus, if in the usual interpreter a variable is bound to any ground

term, it would be bound to g in this abstract interpreter; if the term is not (known to be) ground the variable would be bound to a. We call the set of these tokens the *abstract domain.*

The initial query to the above program is written as p(g,a) since f(a) is a ground term and X is a free variable. The abstract interpreter will contain an abstract version of unification which corresponds to the usual unification. Here the abstract unification attempts to match the call p(g,a) with the head p(f(A),B). Abstract unification is defined as follows:

- Unifying any term with g binds all the variables in the term to g.
- Unifying any term with a binds all the variables in the term to a unless already bound to g.
- Unifying any variable to a term binds the variable to g if the term is ground, otherwise to a.
- Constants are unified as in usual unification.

The unification of p(g,a) and p(f(A),B) results in the substitution $\langle A/g\ B/a \rangle$. The two local variables in the clause C and D are initially free and so are bound to a. This gives the initial substitution of the body as $\langle A/g\ B/a\ C/a\ D/a \rangle$.

The first subgoal in the clause calling the predicate q/1 is completely ground, as is the call to the predicate r/1 since the variable C is unified to a term g(A) where A is g. The variable D on the other hand is unified with g(B) where nothing is known about the groundness of B, hence nothing can be determined about the groundness of D. Hence the groundness of the call to the predicate s/1 cannot be determined.

The execution of the abstract interpreter has correctly inferred that, if the original call is of the form p(g,a), all the calls to the predicates q/1 and r/1 are ground, but the groundness of the call to s/1 cannot be determined. Note, that this does *not* imply that the call certainly involves free variables, even though it does when executed on the usual Prolog interpreter. That is, we can only make statements about the program that reflect the particular abstraction used in the abstract interpreter. Here we have chosen an abstraction where one can only say whether a variable is ground or anything.

To illustrate this point further, consider adding an extra clause to the above program:

```
p(f(A), B) :- q(A), C=g(A), r(C), D=g(B), s(D).
p(h(A), B) :- q(B).
?- p(f(a), X).
```

Note that this addition has no effect on the usual Prolog execution because the head matching unification on the new clause will fail. However, it will succeed on the abstract

interpreter—because information about which constants make up terms has been lost—leading to a nonground call to the predicate q/1. For this program, the abstract interpreter could no longer infer the groundness of all the calls to q/1, only the calls in the first clause. Whether this information is still useful depends on the application. There are three points to be made here:

- An abstraction involves a loss of information and so implies a reduction in the precision of the information an abstract interpreter can infer.
- Despite this, the information is always correct; saying that a variable is bound to a is always correct.
- Similarly, if a variable is bound to g then we can be sure that the variable is always ground.

This abstract domain is not sufficiently expressive to be able to represent variables bound together, or *aliased*. After a unification X=Y, where both are initially bound to a, they are still both bound to a. If subsequently X is bound to g then it is not possible to also bind Y to g and so we cannot infer the groundness of Y. It is still correct to say that Y is bound to a because a represents all Prolog terms not just the nonground ones. This is another source of imprecision in this particular abstract domain.

Abstract interpretation allows us to solve the two problems we stated at the beginning of this section as follows:

- Prolog execution is approximated by considering only abstract properties of the calls. Thus a single abstract call can represent a large set of normal calls which would otherwise have to be treated individually.
- Because the set of abstract calls is smaller than that of normal calls, and often is also finite, it is possible to use conventional techniques for handling nonterminating programs [24].

The foundations of abstract interpretation are given in the work of Cousot and Cousot [16, 17, 18]. This work develops conditions for correct but not always precise abstract interpreters. These conditions apply to the primitives in an interpreter for the language, and are formulated in terms of the relationship between the usual domain over which the interpreter executes and the abstract domain over which the abstract interpreter executes. The foundations are applicable to a standard imperative language, but have been reformulated for functional languages [58] and for Prolog [51, 37, 7]. For a number of papers on the abstract interpretation of declarative languages, see [1].

Abstract interpretation has recently been used in a wide variety of applications in logic programming. These include:

- Mode inference [14, 51, 22, 40].
- Type inference [8, 44, 55].
- Occur check reduction [68].
- Shared variables [79, 34, 57].
- Shared data-structures [9, 55].
- Program specialization [30, 28, 29].

Rather than prove the correctness of each application separately, it is more effective to develop a *framework* for the abstract interpretation of a language. A framework provides some suitable semantics for Prolog parameterized on some primitives implementing the abstraction. Some conditions on the abstraction and these primitives are also specified, and, provided these conditions are satisfied, it is guaranteed that the analyser resulting from instantiating the semantics with the primitives is both correct and terminating. Thus the proof effort involved in developing a number of analysers for different applications is minimized. Ideally, the conditions on the abstraction should be sufficiently general and simple to enable a wide range of different abstractions to be easily proved correct.

In Chapter 5 we develop such a framework for Prolog abstract interpretation. The framework is presented as a Prolog program and this leads directly to an implementation. In Chapters 6 and 7 we develop abstract domains suitable for inferring some of the features we use for describing techniques. In Chapter 6 we describe an abstract domain (and its implementation) for inferring the inclusion and sharing relations which represent the effects of unifications which occur during a clause execution. This domain is similar to the abstract domains developed for detecting shared data-structures but infers more precise information and is also based on a particular concrete description of Prolog which has been instrumented with extra information. In Chapter 7 we sketch an abstract domain which infers type information. This information is novel because it captures certain relationships between recursive data-structures. It is also based on an instrumented concrete description of Prolog.

1.5 The Structure of this Thesis

The rest of this thesis is divided into seven chapters:

- In Chapter 2, we review some recent work on Prolog programming techniques. Our conclusions are that none of this work is suitable as the basis for a more expressive P-frame language, primarily because the techniques are largely syntactic and thus difficult to detect in code.

- In Chapter 3, we develop our own language for describing techniques. The main feature used by this language is the effects of unifications which occur during execution. These effects are expressed in terms of two relations: inclusion and sharing. We demonstrate how our techniques language might be used to enhance APROPOS2, and show some of the limitations of our approach.

- In Chapter 4, we review work on abstract interpretation. We give a brief presentation of the theory, describe some recent applications of abstract interpretation to Prolog and describe the different kinds of semantics used as a basis for abstract interpreters.

- In Chapter 5, we develop a framework for Prolog abstract interpretation. This defines properties that abstract domains must satisfy to guarantee that their analyses are both correct and terminating. The framework is presented as a Prolog program and this leads directly to an implementation.

- In Chapter 6, we develop a simple abstract domain for inferring inclusion and sharing. This domain is similar to the abstract domains developed for detecting shared data-structures, but is more precise in certain circumstances. An implementation of the primitives operating over the abstract domain is described.

- In Chapter 7, we sketch a more complex abstract domain which is capable of representing good quality type information. This enables a broader range of programming techniques to be specified accurately and thus overcomes some of the limitations of our techniques description language. We suggest ways of ensuring that an abstract interpreter based on this domain can be made to terminate.

- In Chapter 8, we review the contributions we have made and suggest some areas for further work.

- In the appendix, we give some example executions of the programs.

1.6 Preliminaries

In this thesis we restrict ourselves to pure Prolog supplemented with a few relatively harmless system predicates (such as $>/2$). We assume two transformations on Prolog which reduce code to a normal form but which do not involve any loss of generality:

- Often programmers bundle together a number of arguments into a single argument using some arbitrary constructor function. Whilst this is occasionally a valid optimization, it makes program analysis difficult because it is necessary to distinguish between these constructor functions and those which perform a more

vital part of the computation. We assume that our code has been preprocessed to remove all constructor functions which appear in every call to any particular predicate and are therefore redundant. This transformation is detailed in [29].

- We introduce explicit unifications to make arguments in the head and subgoals distinct variables. All unifications will be explicit calls to the =/2 predicate such that both sides are variables or one side is a variable and the other a term which contains no multiple occurrences of variables. Structures in the head of the clause become calls to =/2 at the beginning of the clause body. Structures in a subgoal become calls to =/2 just before that subgoal in the clause body.

Most of the Prolog code examples in this thesis is given in this normal form. For clarity, we sometimes give the conventional form as well, side by side, for example:

```
flatten(A, B, C) :- A=[D|E],
    flatten(D, B, F),
    flatten(E, F, C).
flatten(A, B, C) :- A=[], B=C.
flatten(A, B, C) :- B=[A|C],
    A\=[], A\=[_|_].

?- A=[a, [b], c], C=[], flatten(A, B, C).
```

```
flatten([H|T], L-L0) :-
    flatten(H, L-L1),
    flatten(T, L1-L0).
flatten([], L-L),
flatten(X, [X|L]-L) :-
    X\=[], X\=[_|_].

?- flatten([a, [b], c], FL-[]).
```

Here the arbitrary constructor function -/2 has been replaced by two separate arguments and all the head unifications have been moved into the body.

The advantages of adopting this normal form are:

- It enables those unifications that occur in the clause head to be treated in the same way as those in clause body. This simplifies our techniques descriptions.

- It allows predicate calls to specified using only the name of the predicate and a vector of argument variable bindings, instead of having to specify any terms in the arguments as well. This makes comparisons between different predicate calls easier in abstract interpreters.

Chapter 2

Prolog Programming Techniques

2.1 Introduction

In the previous chapter (see Section 1.3) we introduced the notion of Prolog programming techniques. In this thesis we view a technique as a common Prolog practice which is independent of the programming problem, and as a device for decomposing and explaining Prolog code. We aim to provide a language for describing a number of classes of techniques which may be used to enhance APROPOS2's P-frame language used to describe example solutions to programming tasks. The type of techniques we focus on, which we believe are most appropriate to APROPOS2 in the first instance, are those which manipulate data-structures.

We have stated two requirements for the description language for techniques:

- The language must be capable of representing descriptions to different degrees of abstraction, to provide a flexible language for describing algorithms.
- It must be possible to detect the features used to describe techniques, so that instances of techniques may be spotted in student's code.

Here we review some recent work which has a techniques theme and describe the kind of technique description language which is used, and discuss whether it meets the above requirements.

2.2 Prolog Schemata

Gegg-Harrison presents a large collection of *schemata* for various types of Prolog predicate which recurse over lists [32]. The most general schema represents all possible predicates which perform one pass through a list. The schemata are organized into class/instance hierarchies along orthogonal dimensions such as the type of termination

criteria used in the predicate. The claim is that this knowledge provides a useful 'complexity' ordering on a sequence of example predicates shown to a novice programmer. Gegg-Harrison envisages a Prolog tutoring program which alternately presents example programs and then sets problems requiring similar solutions. The user is expected to select and fill in schemata to arrive at these solutions. This tutoring program would also incorporate other modules such as a debugger.

The schemata specify necessary recursive and base clauses with appropriate tests and recursive subgoals. Gegg-Harrison rightly claims that for his limited list-processing subset of Prolog some aspects of the semantics of the program are captured in a purely syntactic way. However, these aspects are heavily biased towards the control flow properties of the programs, rather than the data-flow properties.

Two simple schemata Gegg-Harrison presents are:

```
schema_A([],≪&1≫).
schema_A([H|T],≪&2≫) :-
    <pre_preds(≪&3≫,H,≪&4≫)>,
    schema_A(T,≪&5≫),
    <post_preds(≪&6≫,H,≪&7≫)>.

schema_C([E|T],E,≪&1≫).
schema_C([H|T],E,≪&2≫) :-
    (E \= H),
    <pre_preds(≪&3≫,H,≪&4≫)>,
    schema_C(T,E,≪&5≫),
    <post_preds(≪&6≫,H,≪&7≫)>.
```

Here, ≪&1≫ means any number of arguments and <predicates> means an optional subgoal. (It is not clear why the arguments need to be numbered, nor if there may be a number of subgoals in a single <predicates>).

Both these schemata have a similar structure in that they both recurse down a list. They are different in their termination criteria. Schema_A will always process the entire list, whereas schema_C carries an extra argument which is used to match against each list element and to terminate the recursion. For example, the standard append/3 predicate is an example of schema_A and member/2 an example of schema_C[1]:

```
append([], L, L).
append([H|L], K, [H|M]) :- append(L, K, M).

member(X, [X|_]).
member(X, [_|T]) :- member(X, T).
```

A Prolog intelligent tutoring system (PITS) can present these two schemata as either

[1]Note that it is the second argument of member/2 which corresponds to the first argument of schema_C.

similar examples of decomposing a list, or as different examples of termination. This choice is taken at some higher level when considering what is most relevant for a particular student.

Gegg-Harrison acknowledges that a PITS incorporating such an approach would still need a debugging component. Gegg-Harrison describes a vague idea for debugging programs by "merely" transforming a student's program into some canonical model program using fold/unfold [70], and debugging this "against the class of errors associated with this model program". He suggests this as an alternative to Looi's algorithmic approach.

The schemata are very rigid—they specify in considerable detail the form 'correct' predicates should take. There is perhaps scope for breaking down schemata into low-level parts such as 'base clause' or 'recursive clause with test'. Without this a novice programmer may not be encouraged to understand why each clause is present and not gain an appreciation of the significance of clause ordering. Of course, if lower-level parts were taken as basic, the problem of how to explain how they may be combined needs to be addressed—and this is possibly what Gegg-Harrison was trying to avoid.

A second deficiency is that there is no help given to the novice in inducing the remaining 'output' arguments necessary for correctly filling in schemata for a given programming task. (We quote 'output' because in Prolog there is sometimes no distinction between input and output arguments.)

From the point of view of enhancing APROPOS2, Gegg-Harrison's set of schemata is very limited since it only describes techniques used to recurse over lists. The schemata themselves describe specific instances of techniques; there is no attempt at characterizing the entire class of techniques within a single, general schema.

Gegg-Harrison does not address the issue of detecting instances of techniques in code. It would be possible to detect instances of the techniques described using schemata using a simple syntax matcher. It is, however, difficult to imagine this approach being extended to cover more involved Prolog techniques because Prolog code rarely contains many syntactic features. Consider how a syntax matcher would behave when given a slightly buggy instance of a technique. In this case, it is possible that several schemata almost match the code and it would be difficult to infer any useful information.

2.3 Skeletons and Extensions

Kirschenbaum *et al* develop an approach to structured Prolog programming [43]. The approach they take is to identify a set of basic control flows or *skeletons* for programs

and a set of additions[2] to these skeletons. Program construction is then treated as stepwise enhancement of these skeletons by application of these additions; yielding an *extension*. Separate extensions may be composed into a single program.

For example, their skeleton for traversing a list (which would correspond to schema_A above) is:

```
traverse_n([X|Xs]) :-
    b1(X), traverse_n(Xs).
traverse_n([X|Xs]) :-
    b2(X), traverse_n(Xs).
    ⋮
traverse_n([X|Xs]) :-
    bn(X), traverse_n(Xs).
traverse_n([]).
```

The result of applying their accumulator addition to this skeleton (and restricting to one recursive clause) is:

```
traverse_n_accummulate(As, Total) :-
    traverse_n_accummulate(As, [], Total).

traverse_n_accummulate([A|As], Current, Final) :-
    update(As, Bs, Current, Next),
    traverse_n_accummulate(Bs, Next, Final).
traverse_n_accummulate([], Final, Final).
```

Skeletons closely correspond to Gegg-Harrison's schemata above, and the additions described could also be used in his proposed PITS as examples of how to correctly fill in these schemata. However, stronger claims are also made for the skeleton/enhancement approach to program construction. The authors believe that it both captures the proper way to construct large programs—that is, a problem should be continually transformed into smaller subproblems until these smaller problems can be seen to be extensions of single skeletons—and also provides a framework for automating this construction process.

The combination of Gegg-Harrison's 'structured' teaching by providing ever more complex examples alternating with problems set by providing similar schemata to fill in, and Kirschenbaum's view of correct ways to do this filling in, is quite an attractive way to organize programming teaching. It could be claimed, however, that such a tightly bounded environment for novices may not encourage useful experimentation. Nevertheless, there is a degree of creativity required on the part of the novice when selecting the schema to use, and also it is sometimes necessary to mutate skeletons and

[2]They use the term 'techniques' for these additions. Note that our notion of techniques includes their skeletons as well as their additions.

enhancements to get exactly the solution desired. Whether a student actually gains an understanding of recursion from using such a PITS, rather than simply becoming adept at filling in schemata, can only be determined by experimentation.

Although it is possible to combine additions on top of a particular skeleton, it is not possible to combine two skeletons. Thus a new skeleton is needed if the problem requires that two lists should be traversed simultaneously, and it is not possible to make use of the traverse list example above.

Kirschenbaum's research gives credit to the overall techniques approach. However, from the point of view APROPOS2 it suffers from much the same problems as Gegg-Harrison's work; namely, the clause schema based language is not sufficiently flexible to describe abstract classes of techniques, nor does it support anything other than a simple syntax matcher to allow techniques to be detected.

2.4 A Techniques Editor

An alternative approach to program construction is Bundy's recursive techniques editor [11, 12]. This is a structure editor which allows enhancements to be made to the standard schema for primitive recursion in a way which guarantees that the resulting program will terminate and is well-defined—though not all useful programs may constructed in this way. "Will terminate and is well-defined" means that provided the predicate is called in a correct mode no more than one answer is produced for all inputs.

For example, the initial primitive recursive schema is:

$$\begin{aligned} \mu(\flat, \Phi) &= \nu(\Phi) \\ \mu(\sharp(\Psi, \Theta), \Phi) &= \xi(\Psi, \Theta, \Phi, \mu(\Theta, \Phi)) \end{aligned}$$

where μ is the recursive predicate being defined, $\flat$ and $\sharp$ are the constructor functions and so on. The symbols act as placeholders in the schema. It is necessary to use these symbols because partially instantiated schemata need to be represented and the placeholders distinguished from the instantiated parts. For example, a simple editing command renames one of these placeholders. So if the recursive predicate is called *append*, the user would rename μ with *append* and then be presented with:

$$\begin{aligned} append(\flat, \Phi) &= \nu(\Phi) \\ append(\sharp(\Psi, \Theta), \Phi) &= \xi(\Psi, \Theta, \Phi, append(\Theta, \Phi)) \end{aligned}$$

Note that the schemata are functions. A post-processing stage is needed to translate these into valid Prolog predicates; for example, an extra argument needs to be added to the head of the equations.

More complex commands remove or add placeholders, and change the kind of recursion that the predicate will perform. For example, it is possible to increase the step

size of the recursion. For instance, suppose the user is defining a fibonacci predicate and had reached a stage where the schema is as follows:

$$\begin{aligned} \mathit{fib}(0) &\rightleftharpoons \nu \\ \mathit{fib}(s(N)) &\rightleftharpoons \xi(N, \mathit{fib}(N)) \end{aligned}$$

To make *fib* jump down in steps of 2 rather than one, a new constructor function is added around the term $s(N)$ in the recursive clause, resulting in:

$$\begin{aligned} \mathit{fib}(0) &\rightleftharpoons \nu \\ \mathit{fib}(s(0)) &\rightleftharpoons \xi_1 \\ \mathit{fib}(s(s(N))) &\rightleftharpoons \xi_2(N, \mathit{fib}(N), \mathit{fib}(s(N))) \end{aligned}$$

With this insertion various inputs that would have matched this clause will no longer match. To keep the program well defined the editor creates the extra clause. It is also possible that an additional recursive call is needed in the original clause, hence the editor adds the extra argument to ξ. The user might then use further remove and rename operations to produce the end result:

$$\begin{aligned} \mathit{fib}(0) &\rightleftharpoons s(0) \\ \mathit{fib}(s(0)) &\rightleftharpoons s(0) \\ \mathit{fib}(s(s(N))) &\rightleftharpoons \mathit{fib}(N) + \mathit{fib}(s(N)) \end{aligned}$$

The editor is aimed at a rigorous approach to program construction by forcing the programmer to be aware of the cases and the dependencies between arguments that need to be considered. In Prolog, it is particularly easy to accidentally forget about these though an 'expert' programmer may well resent being force-fed in this manner. An environment which contains the right tools to detect and handle mistakes when they occur may well be preferable to the prescriptive style of the techniques editor.

The techniques editor is not primarily intended as an educational tool for novices. It does not advocate any particular teaching style, though a number of 'helpful hint' opportunities are suggested. The editing commands described do not directly compare with Kirschenbaum's additions since they are at a somewhat lower level. It would, however, be possible to define macros over the basic set of low-level editing commands which would correspond to these additions. For example, to cater for the common case of recursing over lists such a macro would rename ♭ to []/0 and ♯ to ./2. These macros would be somewhat more flexible because it would be possible to specify additions in a more abstract way. For example, instead of a list we could recurse over some arbitrary linear recursive data-structure by not specifying the actual names for the nullary and binary function constructors.

Such a macro language could be the basis for a techniques description language. It might be possible to detect techniques in code by noting each macro call. We believe, however, that a useful set of techniques would require the definition of too large a set of editing macros for a student programmer to deal with. In addition, a student programmer will not always apply the macros in a convenient way and may spend much time undoing previous edits. This approach would require this techniques editor to be incorporated into APROPOS2.

2.5 Discussion

Here we review some of the problems with the above research from the point of view of APROPOS2, and motivate the design for a techniques language given in the next chapter.

Most of the above work is concerned with *constructing* Prolog programs. This has led to a view of techniques which is largely syntactic; that is, techniques are described using clause schemata. We claim that this is not appropriate for our purposes because such schema languages are not sufficiently flexible to describe abstract classes of techniques; that is, syntax is not a sufficiently expressive formalism to describe many techniques. Also techniques described in largely syntactic terms are difficult to detect in code, for the following reasons:

- Prolog lacks syntactic features and those syntactic features that do exist may be labelled ambiguously.
- A syntactic definition of accumulator pairs which captures 'disguised' implementations such as the reverse/2 predicate given in Section 1.2 would be very complex.

The following example also illustrates the problems of 'disguised' techniques. Consider the predicate split/3 which can split a list into two:

```
split([], [], []).
split([H|T], [H|L], M) :-
    split(T, M, L).
?- split([a,b,c], [a,c], [b]).
```

This predicate is unusual in that the second and third arguments are swapped with each recursive call. A simple syntactic definition for the technique used for constructing a list in the head of a recursive predicate would state that the variable in the tail of the ./2 pair should appear in the same argument position as the ./2 pair in the recursive subgoal. This definition would clearly miss the use of this technique in the second and third argument positions of split/3. A simple syntactic analysis of this program would not be able to infer much about these arguments.

We believe that these reasons imply that a syntactic approach fails to capture the fundamentals of techniques which manipulate data-structures. The term 'manipulate' suggests an alternative approach to defining techniques: the only way Prolog manipulates data-structures is by unification. In the following chapter we develop an alternative language for describing techniques based on the effects of unifications occurring during a clause execution.

2.6 Summary

In this chapter, we have

- reviewed some recent work on Prolog programming techniques.
- motivated the need for an improved, non-syntactic technique description language.

Chapter 3

A Description Language for Prolog Techniques

3.1 Introduction

In Chapter 2 we described some approaches to Prolog programming techniques and showed than none of them are suitable for enhancing APROPOS2's P-frame language [46].

The main limitation is that techniques are regarded as essentially syntactic. In this chapter we attempt to overcome these limitations by developing a new approach to Prolog techniques which characterizes techniques in terms of the dynamic effects of unifications which occur during program execution, as well as static feature such as syntax. We show how such techniques might be used in the P-frame language, and show how to detect techniques in students' code provided it is annotated with various information.

Note that the schemata we present here are intended as definitions of classes of techniques not templates used directly as a teaching aid; the language is far too abstract to be of any use to students for program construction.

3.2 Inclusion and Sharing

Before we start to present the techniques description language we need to consider how the effects of unifications are to be specified. We base our techniques description language on two relationships between the variables in clauses which we have named *inclusion* and *sharing*. For the moment we will illustrate these relationships by some example unifications. We give a precise definition in Section 3.10.

Inclusion is intended to capture the idea that unifications are often used to both access parts of a data-structure, and also construct data-structures out of a number of

parts. The inclusion relationship is illustrated by the call:

?- X=[H|T].

After this call we say that X includes T and X includes H; the terms bound to H and T are the same as some part of the term bound to X. Note, however, that inclusion does not specify *which* part of the terms involved are the same. In other words, details of the form of the data-structures involved are not important. We write 'X includes Y' as X$\gg$Y.

Sharing is intended to capture the idea that often parts of a data-structure are used to build another data-structure. The sharing relationship is illustrated by the call:

?- X=[H|T], Y=[H|S].

After this call we say that X shares with Y, because of the variable H appearing in both unifications. Again, no particular part is specified. We write 'X shares with Y' as X$\sim$Y. Lastly, note that a unification:

?- X=Y.

causes X and Y to share, X to include Y and Y to include X.

3.3 Simple Clause Level Techniques

Here we identify a number of classes of simple techniques which are used in individual clauses. It is possible to decompose the way in which many data-structures are processed in a Prolog clause into instances of these technique classes. In later sections, we combine several clause level technique classes into more complex clause level technique classes, and also into technique classes over entire predicates and over several predicates.

3.3.1 The *Sharing* Technique

We first consider how the sharing relation can be used to define techniques. Fresh variables are initially independent of each other so if two variables share then a unification must have occurred which constrained some part of their values to be the same. A statement we can make about the effects of a subgoal is that it causes two variables to share where they previously did not. We write this by annotating the subgoal with the sharing relationship as follows:

subgoal:X$\sim$Y

Here ':' is used to annotate syntactic features, and 'subgoal' refers to both user subgoals and explicit unifications. Note that we assume the code is preprocessed into a normal form (see Section 1.6).

This statement is the basis of our first technique definition called *Sharing*. A technique specifies a number of features which, if present in a clause, allows us to label one or more of the argument positions in the head. The following technique description assigns the labels *Sharing1* and *Sharing2* to two argument positions in a clause:

> Two head argument positions can be labelled *Sharing1* and *Sharing2* if
> X and Y are the variables in those argument positions
> there is a subgoal which makes X share with Y

This is written more concisely as a clause schema which defines a class of techniques:

> head(X:*Sharing1*,Y:*Sharing2*) :-
> subgoal:X∼Y

This uses the annotation introduced above to specify a subgoal which should appear in the clause body. There can be additional subgoals in the body. The variable names X and Y need not be part of the subgoal. They should appear in the head of the clause though they need not be the only variables to appear, nor do they have to appear in this order.

Consider the familiar append/3 predicate[1]:

```
append(A, B, C) :- A=[], B=C.
append(A, B, C) :- A=[D|E], C=[D|F],
   append(E, B, F).
```

```
append([], L, L).
append([H|L1], L2, [H|L3]) :-
   append(L1, L2, L3).
```

In the first clause, assuming the inputs initially do not share, the second and third argument positions form an instance of the *Sharing* technique class. This is because of the unification B=C which causes B and C to share. To illustrate technique instances we will underline the relevant parts of the clause, add the sharing annotations caused by these parts and add the techniques labels to the argument positions of the clause:

append(A, B:*Sharing1*, C:*Sharing2*) :- A=[], B=C:B∼C.

The first argument variable A is not made to share with any other variable during the clause execution, and so no *Sharing* technique instance involves A in this clause.

The *Sharing* technique is simple but too general; often it is possible to label most pairs of argument positions with *Sharing*. The second clause of append/3 is an example of this. To handle this we need more specific technique classes.

3.3.2 The *Up* Technique

Consider again the append/3 predicate. In the second clause the third argument position variable C is first unified with a term [D|F], and subsequently the variable F is

[1] Recall that we will often give some code in our normal form and in a more usual form (see Section 1.6)

passed to another call. A similar pattern is seen for the first argument position variable A which is first unified with the term [D|E]. This pattern illustrates another class of techniques, called *Up* techniques[2].

This technique class is defined using the inclusion relation. We write inclusion relationships by annotating subgoals in clause bodies, just as we did for sharing relationships:

subgoal:X≫Y

This annotation means that a call to this subgoal has the effect of making X include Y.

The instance of *Up* in the third argument of append/3 is illustrated by the following (with the relevant parts of the code underlined):

append(A, B, C:*Up*) :-
  A=[D|E],
  C=[D|F]:C≫F,
  append(E, B, F).

From this, we develop a description of the technique class:

A head argument position is *Up* if
  X is the variable in that position
  there is a subgoal which makes X include another variable Y
  there is another subgoal involving Y

Note that the description does not specify that the subgoal involving Y must be recursive. The description of the class is more conveniently stated as a schema:

head(X:*Up*) :-
  subgoal:X≫Y &
  subgoal(Y)

In our notation, & is used to join different subgoal specifications. There is no ordering implied by &; the subgoals may be in any order but they are *different* subgoals. This means that the third argument of the second clause of append/3 could still be given the *Up* label if the clause were rewritten as follows:

append(A, B, C:*Up*) :-
  A=[D|E],
  append(E, B, F),
  C=[D|F]:C≫F.

This rewritten version is still a plausible implementation of append/3, though it can enter an infinite loop with calls of certain modes[3]. This implies that we do sometimes

[2]Because one of their uses is to build data-structures on the way up out of a call.
[3]A mode is the instantiation state of an argument of a predicate call.

need to specify an ordering of the subgoals in a technique[4]. For this we introduce a new construct to join different subgoal specifications &≻ which is read 'and then'.

We can define another (sub)class of techniques called *PrologUp* which is just like *Up* but with this additional ordering constraint:

```
head(X:PrologUp) :-
    subgoal:X≫Y &≻
    subgoal(Y)
```

The original version of the second clause of append/3 is an instance of this technique in its third argument, but the rewritten version, with the unification after the recursive call, is not. Note also that the first argument instantiates *PrologUp* as well:

```
append(A:PrologUp, B, C) :-
    A=[D|E]:A≫E,
    C=[D|F],
    append(E, B, F).
```

Note that the first and third argument positions of append/3 are typically called with different modes but both are instances of the *Up* technique.

The pattern of dependencies between the variables in the above definitions may be generalized to give the description of a class of *Up* techniques:

An argument position is *Up* if
- X_0 is the variable in that argument position
- there is a call which makes X_0 include X_1
- there is another call which makes X_1 include X_2
- ⋮
- there is another call which makes X_{n-1} include X_n
- there is another call involving X_n

or in the schema form:

```
head(X0:Up) :-
    subgoal:X0≫X1 &
    subgoal:X1≫X2 &
        ⋮
    subgoal:Xn-1≫Xn &
    subgoal(Xn)
```

The append/3 example is an instance of this where n is 1, X_0 is C and X_1 is F. This 'general' definition above captures the important characteristics of the *Up* techniques class. There is scope for defining a hierarchy of different techniques which are subclasses of *Up* or combinations of several *Up* techniques. These may be based on:

[4]Note that in the wider context of logic programming in general there exist languages in which the ordering of subgoals is not important at all.

- The order of the calls.
- The effect of other variable dependencies not specified in the definition.
- The mode of the argument position.

One such subclass is the degenerate case where n is 0; the schema describing this class is:

head(X_0:*Up*) :-
  subgoal(X_0)

3.3.3 The *Down* Technique

Consider the following predicate which searches a graph building in its third argument a list of the nodes which have been visited:

```
search(A, B, C) :- A=B.
search(A, B, C) :-
   arc(A, D),
   not_member(D, C),
   E=[D|C],
   search(D, B, E).

search(E, E, _).
search(N, E, V) :-
   arc(N, O),
   not_member(O, V),
   search(O, E, [O|V]).
```

In the second clause the third argument position variable C is involved in the unification E=[D|C], and then E is passed into the call search(D, B, E). This pattern illustrates another class of techniques, called *Down* techniques.

The instance of *Down* in the third argument search/3 is illustrated by the following:

search(A, B, C:*Down*) :-
  arc(A, D),
  not_member(D, C),
  E=[D|C]:E≫C,
  search(D, B, E).

From this we develop a description of the class of *Down* techniques:

An argument position is *Down* if
  X is the variable in that argument position
  there is a call which makes another variable Y include X
  there is another call involving Y

and written in schema form:

head(X:*Down*) :-
  subgoal:Y≫X &
  subgoal(Y)

As with *Up*, this may be generalized:

head(X_0:*Down*) :-
  subgoal:$X_1 \gg X_0$ &
  subgoal:$X_2 \gg X_1$ &
    ⋮
  subgoal:$X_n \gg X_{n-1}$ &
  subgoal(X_n)

As with the *Up* case, the actual technique used in the code is more precisely described if an ordering is specified for the subgoals. For this we introduce the new techniques class *PrologDown*:

head(X_0:*PrologDown*) :-
  subgoal:$X_1 \gg X_0$ &$\succ$
  subgoal:$X_2 \gg X_1$ &$\succ$
    ⋮
  subgoal:$X_n \gg X_{n-1}$ &$\succ$
  subgoal(X_n)

Another sub-class of *Down* is the case where n in the general definition is 0:

head(X_0:*Down*) :-
  subgoal(X_0)

Note that this is indistinguishable from the corresponding *Up* case; so it may be possible to assign a number of labels to an argument position.

3.4 Complex Clause Level Techniques

More complex technique classes can be seen as combinations of the simple techniques *Down* and *Up*. We define a general class of technique which we call *dependency chains* as a combination of a single *Up* and a single *Down* technique with the following schema which annotates arguments with the labels *DepDown* and *DepUp*:

head(X_0:*DepDown*, X_n:*DepUp*) :-
  subgoal:$X_1 \gg X_0$ &
  subgoal:$X_2 \gg X_1$ &
    ⋮
  subgoal:$X_n \gg X_{n-1}$

An example of an instance of a dependency chain technique is the palindrome program palin/3 which relates a list to its palindromic list:

```
palin(A, B) :- C=[], palin(A, C, B).        palin(L, P) :- palin(L, [], P).

palin(A, B, C):- A=[], B=C.                 palin([], P, P).
palin(A, B, C) :- A=[D|E], C=[D|G],         palin([H|T], P0, [H|P]) :-
   F=[D|B], palin(E, F, G).                    palin(T, [H|P0], P).
```

It can be used as follows:

```
?- palin([a,b], X).

X=[a,b,b,a]
```

The second argument position in the third clauses may be given a label *DepDown* and the third argument position *DepUp*, as follows:

```
palin(A, B:DepDown, C:DepUp) :-
   A=[D|E],
   C=[D|G]:C≫G,
   F=[D|B]:F≫B,
   palin(E, F, G):G≫F.
```

Note that ordering of the subgoals in this clause does not correspond to the ordering of the subgoal specifications in the technique schema. This is fine, because we are using & and not &≻, but it does make the match hard to follow. Here X_0 is B, X_1 is F, X_2 is G and X_3 is C.

3.4.1 Accumulator Pairs

There are few examples of general dependency chain techniques in everyday code but instances of subclasses crop up regularly. One such sub-technique is that of *accumulator pairs*, as used in the standard 'quick' reverse list predicate:

```
reverse(A, B) :- C=[], qrev(A, C, B).       reverse(L, R) :- qrev(L, [], R).

qrev(A, B, C) :- A=[], B=C.                 qrev([], R, R).
qrev(A, B, C) :- A=[D|E], F=[D|B],          qrev([H|T], R0, R) :-
   qrev(E, F, C).                              qrev(T, [H|R0], R).
```

Accumulators seem to have developed quite a specific folklore meaning though there does not appear to be any precise definition. The data-flow corresponds to the dependency chain technique above. In addition to this, there are a number of pragmatic conditions:

- The call mode of the *DepDown* argument position should be instantiated and that of the *DepUp* argument position should be free[5].

[5]It might be more accurate to go further and insist that the input mode of the *DepDown* argument position should be the base case of some recursive data-structure, if such information is known.

- The subgoals in the clause should occur in a particular order.

The definition language we have used for techniques so far cannot express the first of these conditions. To do so, we allow mode annotations in addition to the technique label in the head. The defining schema for a class of accumulator pair techniques[6] which assigns the *Acc* and *AccResult* labels is:

head(X_0:*call*(i):*Acc*, X_n:*call*(f):*exit*(i):*AccResult*) :-
  subgoal:$X_1 \gg X_0$ &≻
  subgoal:$X_2 \gg X_1$ &≻
  ⋮
  subgoal:$X_n \gg X_{n-1}$

Mode annotations may apply to predicate calls *call*(_) and exits *exit*(_), and may specify whether the argument is free f, ground g (that is, fully instantiated) or instantiated i.

The second clause of the qrev/3 contains an instance of the accumulator pair techniques as follows:

```
qrev(A, B:Acc, C:AccResult) :-
   A=[D|E],
   F=[D|B]:F≫B,
   qrev(E, F, C):C≫F.
```

3.4.2 Difference Structures

Another sub-technique of dependency chains are difference structures as used in the second and third arguments of the following qsort/3 predicate:

```
quicksort(A,B) :- C=[], qsort(A,B,C).

qsort(A, B, C) :- A=[D|E],
   partition(D, E, F, G),
   H=[D|I], qsort(F, B, H),
   qsort(G, I, C).
qsort(A, B, C) :- A=[], B=C.

partition(A, B, C, D) :-
   B=[],C=[],D=[].
partition(A, B, C, D) :-
   B=[E|F], C=[E|G], A @≥ E,
   partition(A, F, G, D).
partition(A, B, C, D) :-
   B=[E|F], D=[E|G], A @< E,
   partition(A, F, C, G).
```

```
quicksort(L, S) :- qsort(L, S, []).

qsort([H|T], S, S0) :-
   partition(H, T, Ls, Gs),
   qsort(Ls, S, [H|S1]),
   qsort(Gs, S1, S0).
qsort([], S, S).

partition(_, [], [], []).
partition(X, [H|T], [H|Ls], Gs) :-
   X @≥ H,
   partition(A, F, Ls, Gs).
partition(X, [H|T], Ls, [H|Gs]) :-
   X @< H,
   partition(A, F, Ls, Gs).
```

[6] Note that this definition does not actually cover all instances of accumulator pairs, only those where the relationship between the variables can be captured by inclusion.

Difference structures are a novel feature of Prolog (and its sister languages) exploiting the logical variable by allowing data-structures to be built in a flexible way—in particular, a data-structure is built 'in reverse' top-down rather than bottom-up. In the first clause of the qsort/3 predicate the output variable S is partly instantiated by the first recursive call, and completed by the second recursive call. There is, however, no precise definition of difference structures in the literature, just a set of example predicates.

It is possible to identify a 'typical' subclass of difference structure techniques which may be seen as 'reversed accumulators' used to efficiently concatenate lists. The qsort/3 predicate is an instance of this, as is flatten/3:

```
flatten(A, B) :- C=[], flatten(A, B, C).

flatten(A, B, C) :- A=[D|E],
    flatten(D, B, F),
    flatten(E, F, C).
flatten(A, B, C) :- A=[], B=C.
flatten(A, B, C) :- B=[A|C],
    A\=[], A\=[_|_].
```

```
flatten(L, FL) :- flatten(L, FL, []).

flatten([H|T], L, L0) :-
    flatten(H, L, L1),
    flatten(T, L1, L0).
flatten([], L, L).
flatten(X, [X|L], L) :-
    X\=[], X\=[_|_].
```

Types may be specified in schemata in a similar way to modes, using *type*(_). The schema for typical difference lists techniques class, which assigns the labels *TypicalDL-Down* and *TypicalDLUp* to two argument positions, is:

```
head(X0:TypicalDLDown, X3:type(list):call(f):exit(i):TypicalDLUp) :-
    dirrecuser:X3≫X2 &≻
    [ unify:X2≫X1 &≻]
    dirrecuser:X1≫X0
```

Note that the call mode of the *TypicalDLDown* argument is not in general instantiated. This schema introduces some more notation:

- To allow the kind of subgoal to be specified special tokens are used in place of 'subgoal'. In the above, we have used 'dirrecuser' for directly recursive user subgoals (that is, subgoals with the same name as the predicate they appear in) and 'unify' for simple unifications. Other useful types are 'user' for user subgoals and 'recuser' for recursive user subgoals which may not be directly recursive.
- To allow optional subgoal specifications we use square brackets. Note that we assume the indices of variables are adjusted accordingly.

The first clause of flatten/3 instantiates this technique as follows:

```
flatten(A, B:TypicalDLUp, C:TypicalDLDown) :-
    A=[D|E],
    flatten(D, B, F):B≫F,
    flatten(E, F, C):F≫C.
```

It is this 'reversed order' property which characterizes difference structures in general. The schema for difference structures is the same as dependency chains with a particular subgoal ordering—one that is the reverse of the natural ordering given in the dependency chain definition. The defining schema for difference structures which assigns the *DsDown* and *DsUp* labels is:

head(X_0:*DsDown*, X_n:*exit*(*i*)*DsUp*) :-
  subgoal:$X_n \gg X_{n-1}$ &$\succ$
    ⋮
  subgoal:$X_2 \gg X_1$ &$\succ$
  subgoal:$X_1 \gg X_0$

An example of this more general form is the following program which is a Prolog solution to the Dutch flag problem[7] which uses a novel version of difference structures to efficiently concatenate all the lists together in a single recursive call [69]:

```
dutch(A, B) :-
   C=D, E=F, G=[],
   dl(A, B, C, D, E, F, G).

dl(A, B, C, D, E, F, G) :-
   A=[H|I], H=r(J), B=[H|K],
   dl(I, K, C, D, E, F, G).
dl(A, B, C, D, E, F, G) :-
   A=[H|I], H=w(J), D=[H|K],
   dl(I, B, C, K, E, F, G).
dl(A, B, C, D, E, F, G) :-
   A=[H|I], H=b(J), F=[H|K],
   dl(I, B, C, D, E, K, G).
dl(A, B, C, D, E, F, G) :-
   A=[], B=C, D=E, F=G.
```

```
dutch(Xs, RWB) :-
   dl(Xs, RWB-WB, WB-B, B-[]).

dl([r(X)|Xs],[r(X)|R]-R1,W,B) :-
   dl(Xs, R-R1, W, B).

dl([w(X)|Xs],R,[w(X)|W]-W1,B):-
   dl(Xs, R, W-W1, B).

dl([b(X)|Xs],R,W,[b(X)|B]-B1) :-
   dl(Xs, R, W, B-B1).

dl([], R-R, W-W, B-B).
```

In the first clause of dl/7, the second and third argument positions form an instance of the difference structure technique as follows:

dl(A, B:*DsUp*, C:*DsDown*, D, E, F, G) :-
  A=[H|I], H=r(J),
  B=[H|K]:B$\gg$K,
  dl(I, K, C, D, E, F, H):K$\gg$C.

Note that the fourth and fifth and the sixth and seventh argument position pairs instantiate the difference structure technique in a similar way.

[7] That is, sort a list each element of which is tagged with either red, white or blue so that all the red elements come first, then the white and then the blue, but preserving the original relative ordering of the elements of each colour.

3.5 Other Combinations of Clause Level Techniques

It is possible to combine *Down* and *Up* (and other) techniques into other commonly occurring combinations. For example, in the second clause of a maplist/2 predicate:

```
maplist(A, B) :-
    A=[], B=[].
maplist(A, B) :-
    A=[C|D], B=[E|F],
    map(C, E), maplist(D, F).
```

both arguments are instances of the *Up* technique class. We can also define a technique which captures some of the features of such 'mapping' predicates:

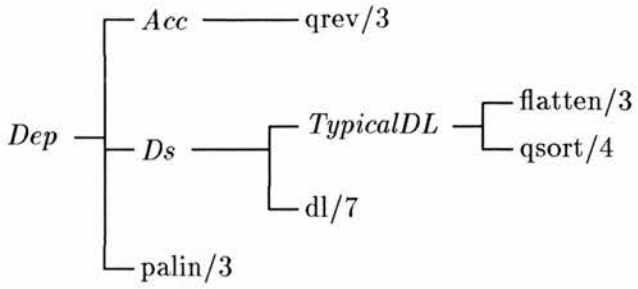
```
head(I:MapOne, O:MapTwo) :-
    subgoal:I≫IH, I≫IT &
    subgoal:O≫OH, O≫OT &
    subgoal(IH, OH) &
    subgoal(IT, OT)
```

3.6 A Classification of Clause Level Techniques

We have suggested throughout that for all the techniques we have defined there is scope for defining various sub-techniques based on a number of orthogonal dimensions. The dimensions are:

- Dataflow as specified by inclusion and sharing relations. We have identified four types; *Sharing*, *Down*, *Up* and dependency chains *Dep*.
- Call and exit modes as specified by mode annotations.
- Type information as specified by type annotations. This ranges from the abstract structural relations such as 'binary tree' or 'linear' through to concrete instances of these like 'list'. Also desirable are non-structural types like 'ordered list'.
- Subgoal ordering as specified by &≻.
- The type of subgoals. That is whether a subgoal is a simple unification, a user subgoal, a recursive user subgoal or a directly recursive user subgoal.

This classification can take the form of a multiple inheritance hierarchy from the most general or abstract techniques at the top, to concrete Prolog instances of techniques at the bottom. The following tree is a part of this hierarchy placing some of the techniques we have described in previous sections, together with example predicates at the leaves:



This tree is limited to the techniques we have described involving a combination of one *Down* and one *Up* technique. At the root of the tree are dependency chains (*Dep*) which only specify some inclusion relationships. The palin/3 program is an example instance. Below dependency chains are general difference structures (*Ds*) whose only extra condition is that the subgoals are in a particular order. The dl/7 predicate is an example instance of the difference structures technique. A sub-technique of difference structures are 'typical' difference lists (*TypicalDL*), with examples including flatten/3 and qsort/3. Accumulator pairs (*Acc*) are also placed below dependency chains since they specify a particular subgoal ordering (different from difference lists) and mode. The program qrev/3 contains an example of accumulator pairs.

This hierarchy shows how these techniques are related. For example, it explains the similarity between accumulator pairs and difference structures. An experienced Prolog programmer will already understand this relationship, but a student may benefit from having these techniques described in this way. Note, however, we do not claim that this hierarchy can be used as a device for organizing a Prolog teaching session, in the same way as Gegg-Harrison [32] uses his techniques hierarchies (see Section 2.2). The dependency chain technique is too abstract and uncommon to be of great direct value during teaching.

3.7 Predicate Level Techniques

The schemata above define techniques in clauses. They are used to label argument positions in the head of the clause, but they take no account of any interactions between different clauses in a program. At this predicate level it is possible to specify combinations of clause level techniques. For example, a standard instance of accumulators would involve a degenerate instance of *Acc* and *AccResult* in the base clause, and some general instance in recursive clauses.

For this we need to group several clause schemata into a single predicate schema by specifying a set of clause techniques:

AccPairPred:{
head(X_0:*Acc*, X_1:*exit*(i):*AccResult*) :-
  unify:$X_1 \gg X_0$
⋮
head(X_0:*Acc*, X_n:*exit*(i):*AccResult*) :-
  subgoal:$X_1 \gg X_0$ &≻
  subgoal:$X_2 \gg X_1$ &≻
    ⋮
  recuser:$X_n \gg X_{n-1}$
⋮
}

We use vertical dots '⋮' if we need to specify that more similar clauses can appear (just as we have been doing with subgoals).

Similarly, it is possible to specify sets of clause techniques which are buggy or odd in some way.

3.8 Enhancing P-frames with Techniques

The aim of our techniques classification is to enhance APROPOS2's P-frame language. Here we describe the kind of benefits we believe our techniques description language will provide.

Simply having the techniques language available provides another feature with which APROPOS2 might guide its matching process. So, for example, we can add a slot to the P-frame description of append/3 stating that it should be possible to label the first argument with the *Up* technique label and the second and third arguments with *DepDown* and *DepUp*.

Our techniques language is more powerful than that used by APROPOS2. For example, we have defined techniques such as accumulator pairs and difference structures. This is the only attempt known to us at defining precisely what is meant by these techniques. Having these definitions available allows implementations of algorithms to be specified in P-frames with considerable accuracy. One example of this is using the 'classic' difference list technique definition in the P-frame for the quick sort algorithm. This appears to be outside the scope of APROPOS2.

We expect that the benefits of using our techniques language will become more pronounced as larger programs are described using P-frames.

Lastly, we suggest another benefit that our techniques classification may provide. An important problem with P-frames is that there is no way of expressing the similarities between different predicates. Consider the problem of flattening a nested list. We have given one solution to this problem above using a difference list:

```
flatten(A, B) :- C=[], flatten(A, B, C).

flatten(A, B, C) :- A=[D|E],
    flatten(D, B, F),
    flatten(E, F, C).
flatten(A, B, C) :- A=[], B=C.
flatten(A, B, C) :- B=[A|C],
    A\=[], A\=[_|_].
```

```
flatten(L, FL) :- flatten(L, FL, []).

flatten([H|T], L, L0) :-
    flatten(H, L, L1),
    flatten(T, L1, L0).
flatten([], L, L).
flatten(X, [X|L], L) :-
    X\=[], X\=[_|_].
```

It is also possible to use accumulator pairs by reversing the order of the subgoals in the recursive clause:

```
flatten(A, B, C) :- A=[D|E],
    flatten(E, F, C),
    flatten(D, B, F).
```

```
flatten([H|T], L, L0) :-
    flatten(T, L1, L0),
    flatten(H, L, L1).
```

With P-frames the similarities between these two alternative solutions cannot be captured. Each recursive clause will have its own separate description. In our techniques classification, accumulator pairs and difference lists are both instances of a more general class of techniques called dependency chains. There are situations where it is pointless for the debugging component of a PITS to try to distinguish between these two implementations. For example, if the student made some error in writing this clause, the fact that there is an extra argument would suggest that one of the above techniques was what the student intended, but there may be no way of telling which. APROPOS2 would try to infer which of these implementations was intended by applying its usual score function. This is a situation which is better handled in some other component of a PITS which has some knowledge about the student. By highlighting the similarities between different solutions we have moved the emphasis away from an *ad hoc* score function as used by APROPOS2.

3.9 Spotting Techniques in Students Code

The schemata we have developed for describing various classes of Prolog techniques may be used to spot instances of techniques in Prolog code—that is to infer clause technique labels for the argument positions in clause heads and predicate technique labels for entire predicates.

Assuming that the code has been annotated with all the possible inclusion and sharing relationships formed by each subgoal, technique instances may be spotted by a straightforward syntax matching between code and schema. We have implemented such a matching program. See the appendix for some examples.

Often any particular argument position may be given labels corresponding to a number of techniques. Such ambiguity may be handled by picking a technique label

corresponding to the most specific technique class (that is, one that specifies the most features) and by attempting to arrive at a consistent labelling for the argument position across all the clauses in the predicate.

We now consider where the annotation information is to come from. In the context of Looi's APROPOS2 program the student cannot be expected to annotate code, so the information must be inferred by some automatic means. Some of the information is syntactic. It is straightforward to build an analysis program that can spot such static features; for example, recursive subgoals may be detected by constructing a call graph of the program. Other information about the program's execution is not so easy to infer; the techniques used to infer this information are the subject of much of the rest of this thesis.

3.10 Inclusion and Sharing Revisited

Inclusion and sharing are intended to capture the effects of unifications; it is by using unification that Prolog programs manipulate data-structures. It is natural that descriptions of data-structure manipulating techniques should specify patterns of unifications.

Until now we have only introduced the inclusion and sharing relations informally by describing the effects of some example unifications. To detect techniques we need to be able to annotate code with inclusion and sharing relations, and a precursor to this is a precise description of inclusion and sharing. We can define inclusion and sharing more precisely, as follows:

- A variable X includes another Y if Y's term is equal to a subterm of X.
- A variable X shares with another Y if Y's term contains a subterm which is equal to a subterm of X.

These definitions assume some notion of term equality. Here we specify a suitable notion of term equality such that the techniques it enables us to describe capture our intuitions about those techniques.

To illustrate the problems, consider the following pair of predicates:

```
p(X, Y) :- X=f(a), Y=f(a).
q(X, Y) :- X=f(a), Y=X.
```

If we take the 'standard' notion of term equality (the equality that unification is based on) both these clauses can be assigned the *Sharing* technique in the following way:

```
p(X:Sharing1, Y:Sharing2) :- X=f(a), Y=f(a):X~Y.
q(X:Sharing1, Y:Sharing2) :- X=f(a), Y=X:X~Y.
```

We claim that this is not an intuitive technique labeling because in predicate p/2 no unification occurs between X and Y (or any part of X and Y). In p/2 the terms bound to X and Y *happen* to have the same appearance, which is enough for the standard equality to regard them as equal, even though there may not have been any intention on the part of the programmer for them to be equal. In q/2 a unification between X and Y does occur and so the technique labeling is reasonable.

A solution to this problem is to have some representation which enables us to distinguish between different occurrences of terms. For this we use *indexed* terms where each function symbol is annotated with an index which is generated for each clause execution. This is similar to the way the variables in a clause need to be renamed with each clause execution.

We define the notion of indexed term equality as follows. Firstly, it requires the usual reflexivity and transitivity rules:

$$X = Y \Rightarrow Y = X$$
$$X = Y \wedge Y = Z \Rightarrow X = Z$$

The variables $X, Y, \ldots$ are meta-variables ranging over terms. In addition, it requires that indexed terms are equal only if all their arguments are equal and the function symbols have the same index:

$$f_i(X_1, \ldots, X_n) = f_i(Y_1, \ldots, Y_n) \Leftrightarrow X_1 = Y_1, \ldots, X_n = Y_n$$

for all function symbols f_i/n in the program. Lastly, we need to take into account the unifications which have occurred so far during the execution of the program. These are regarded as extra axioms. For example, at the end of the execution of the p/2 predicate the set of unifications is:

$\{ \mathrm{X}=\mathrm{f}_i(\mathrm{a}_j), \mathrm{Y}=\mathrm{f}_k(\mathrm{a}_l) \}$

and for q/2 it is:

$\{ \mathrm{X}=\mathrm{f}_i(\mathrm{a}_j), \mathrm{Y}=\mathrm{X} \}$

Here the indices are constructed during execution. This theory of indexed term equality allows us to infer that in q/2's execution X shares with Y after the call to X=Y because X's term $\mathrm{f}_i(\mathrm{a}_j)$ is (trivially) equal to Y's term $\mathrm{f}_i(\mathrm{a}_j)$. In p/2 we cannot infer that X shares with Y. Thus it correctly captures our intuitions.

An important property of this term equality is that it is independent of the instantiation state of the variables involved in each of the unifications. For example, we can label both the following predicates with *Sharing*:

r(X:*Sharing1*, Y:*Sharing2*) :- X=f(a), Y=X:X~Y, Y=f(a).
s(X:*Sharing1*, Y:*Sharing2*) :- X=f(a), Y=f(a), Y=X:X~Y.

despite the different subgoal ordering in each predicate. This is important because it allows techniques to be detected in a way independent of the instantiation state of the variables in the clause; for example, even when calls are ground. This allows us to infer that the first and third arguments in the second clause of append/3:

```
append(A, B, C) :- A=[], B=C.
append(A, B, C) :- A=[D|E], C=[D|F], append(E, B, F).
```

share regardless of the mode of the call to append/3

Finally, note that our notion of indexed terms is very similar to the way that terms are implemented in real Prolog systems. In a Prolog implementation a term is a graph where the nodes are records labelled with function symbols and the edges are pointers. Different records may be labelled with the same function symbol. Two terms are only equal if they are implemented with the same node in the graph. This equality may be called *pointer equality.* Unification operates over these graphs by redirecting pointers. Note that the usual implementation of unification only performs a check if the two terms being unified are ground; that is, it does not force two ground terms to be pointer equal. Thus this representation does not capture the same equality as that of indexed terms. In Chapter 6 we describe a unification algorithm over indexed terms which implements indexed term equality.

3.11 Limitations

We have developed a language for describing a wide range of techniques both in clauses and in predicates. The definitions are based on two relationships, inclusion and sharing, which allow some of the ways in which data-structures are manipulated to be conveniently specified. In addition to these relations a number of other, mainly static, features can be used to define techniques. For the purposes of defining the techniques that are of most immediate interest, namely data-structure manipulating techniques such as accumulator pairs and difference structures, we have used type and mode information and specified the kind of subgoals and their order.

Other techniques may well require more features to be included in the techniques language. For example:

- If a particular sub-technique of *Up* were required which involved skipping along a list two elements at a time, it is necessary to specify that the unification has the form 'X=[_,_|Y]'.

- It would be useful to allow the inclusion and sharing relations to specify the type of the subterms involved. A list matching unification such as:

  X=[H|T]

may then be more accurately described by the subgoal specification:

unify:X$\gg$H:*type*(*element*), X$\gg$Y:*type*(*list*)

- One large omission from the current language is negation; it is impossible to state that some feature is not present. This may be necessary for defining certain techniques, especially those describing buggy techniques such as 'missing base case'.

Inclusion and sharing only capture simple structural relationships between variables. Although this is sufficient for describing certain Prolog techniques there are many techniques for which this is not expressive enough. To illustrate, we give two example predicates. Firstly, consider again the quick sort program:

```
quicksort(A, B) :- C=[], qsort(A, B, C).

qsort(A, B, C) :- A=[D|E],
    partition(D, E, F, G),
    H=[D|I], qsort(F, B, H),
    qsort(G, I, C).
qsort(A, B, C) :- A=[], B=C.

partition(A, B, C, D) :-
    B=[], C=[], D=[].
partition(A, B, C, D) :-
    B=[E|F], C=[E|G], A @≥ E, partition(A, F, G, D).
partition(A, B, C, D) :-
    B=[E|F], D=[E|G], A @< E, partition(A, F, C, G).
```

We have seen how it is possible to label the second and third arguments of the first clause of qsort/3 with the difference structure technique. The first argument is more difficult to label. We can assign the *Up* technique because of the following match:

```
qsort(A:Up, B, C) :-
    A=[D|E]:A≫E,
    partition(D, E, F, G),
    H=[D|I], qsort(F, B, H),
    qsort(G, I, C).
```

Intuitively, this does not capture the effect of the first argument because it ignores the two calls to qsort/3 which further process the input list. A technique description that would better capture this is the following:

```
head(X:Split) :-
    subgoal:X ▷Y, X ▷Z &
    subgoal(Y) &
    subgoal(Z)
```

which introduces a new *partof* relation $\rhd$. Partof is intended to have a similar meaning to inclusion; namely, that a data-structure is in some sense smaller than another. Whereas it is possible to give inclusion a precise meaning in terms of equalities between terms, the semantics of partof are more subjective. In the partition/4 predicate for example, it is clear that the output lists are both sublists of the input list but they are not equal to substructures of it, and so we claim that, after the call to partition/4, the partof relation should hold between the input and outputs.

A version of the partof relation would need to be defined for each data-structure type being compared. For example, in a program where sets were implemented as binary trees the partof relation could be either the subset relation or a subtree relation.

Clearly, any attempt at precisely defining partof would involve a considerable amount of type information. One approach is to extend the sharing and inclusion relations with type information as we mentioned above. For example, we could define a partof relation specific to the qsort/3 example above by extending the sharing relation as follows: 'X shares with Y and the parts that share are not lists'[8].

This approach is a little more flexible but is still limited. For example, it could be claimed that an important part of the *Split* technique as used in qsort/3 is that each element on the input list is used exactly once. Similar information is required to define the relationship between the input and output of predicates such as maplist/3; namely, that they must be the same length. In Chapter 7 we develop a type language in which it is possible to specify such relationships.

The second example shows that some techniques involve no explicit type information or equalities between terms. Closure/2 is a definition of a transitive closure predicate in a graph defined implicitly by the predicate arc/2:

```
closure(A, B) :- A=B.
closure(A, B) :- arc(A, C), closure(C, B).

arc(a,b).
arc(a,c).
arc(b,c).
   ⋮
```

Consider the first argument. It can be seen to be very similar to the first argument of append/3; that is a use of the *Up* technique. Both are traversing some data-structure and then performing some more computation (in both cases a recursive call). The difference is that the data-structure closure/2 is traversing is implicit in the arc/2 predicate, whereas that in append/3 is implemented as a Prolog list. The closure/2 data-structure is not manipulated using unification and so no inclusion or sharing relationships are formed between the variables. It is therefore not possible to assign

[8]This assumes that the input is not nested.

the *Up* technique.

To overcome this problem it is necessary to extend the definition of inclusion. One possible approach would be to regard inclusion as the transitive closure of the 'step' relation of some abstract data-type which the programmer is using implicitly in the code. However, techniques based on such extended definitions would be impossible to automatically detect in code unless extra annotations were provided by some means.

3.12 Summary

In this chapter, we have

- developed a language for describing classes of many common Prolog techniques that manipulate data-structures using inclusion and sharing relationships. This language allows us to formally define techniques such as accumulator pairs and difference structures. Previously such techniques have been characterized only by examples and simple clause schemata.
- placed some abstract descriptions of common techniques within a hierarchy highlighting their similarities and differences.
- suggested how techniques might be used with Looi's P-frame descriptions of predicates.
- suggested how techniques may be detected in Prolog code.
- motivated the need for program analyses to infer the features used to describe techniques.
- shown some of the limitations of our approach.

Chapter 4

Abstract Interpretation in Logic Programming

4.1 Introduction

In Chapter 3 we developed a classification of various Prolog programming techniques suitable for enhancing APROPOS2's P-frame language and showed how it is possible to detect the use of techniques in Prolog code provided that certain information about that code is available. We use analysis methods based on abstract interpretation to infer this information. We informally introduced abstract interpretation in the introduction chapter (see Section 1.4). In this chapter we present more precisely the ideas behind abstract interpretation and review some of the applications of abstract interpretation in the field of logic programming, especially for Prolog. In subsequent chapters we develop specific analysis methods.

4.2 Abstract Interpretation Basics

Abstract interpretation is a principled approach to inferring properties of a program's execution by simulating that execution using an interpreter which computes over some abstraction of the program's usual, concrete, domain and which collects the information of interest during the execution. An abstraction is used so that:

- the properties of interest are more explicit;
- a whole class of queries may be specified by one abstract predicate call;
- and to ensure that, unlike the concrete interpreter, the abstract interpreter terminates for all possible programs and queries.

The cost of abstracting is precision; an abstract interpreter cannot be expected to exhibit the exact behaviour of the normal interpreter, only to approximate it. Note

that the properties inferred can be correct without being precise. For example, for some program it might be possible to guarantee that some property holds, whereas for other programs it may only be possible to infer that this property *may* hold. An extreme case would be a *don't know* result which is completely correct but totally imprecise. The possibility of inferring imprecise information should not be remarkable since many properties of program states will not be decidable.

We informally introduced abstract interpretation in the introduction chapter (see Section 1.4). Here we outline a formalization of abstract interpretation.

We require a concrete semantic function for Prolog which will be abstracted to form an abstract semantic function. Such a concrete semantic function could have the form $Prolog_P : C \rightarrow C$ where P is a program and each member of C is a set of variable substitutions[1]. This roughly corresponds to a conventional concrete semantic function for Prolog; *Prolog* is a mapping from a set of substitutions (one corresponding to each of the queries which will be run on the program[2]) to a set of substitutions (one for each successful resolution of those queries).

The purpose of abstract interpretation, however, is to infer some property of the execution of a program; the conventional variable substitution output of the program is not usually of interest. Rather, it is the details of the execution of the program that is of interest. Accordingly, the concrete interpreter *Prolog* should provide a description of the states reached during the execution of P. Depending on the application, there is plenty of scope for deciding the exact nature of an execution state. For example, with a semantic function based on SLD resolution an execution state would be the AND/OR tree constructed so far, or, for a particular Prolog implementation, the image of the executing Prolog system. Any interesting property of the execution of a program, such as the groundness of the variables or the height of the recursion stack, should be inferable from the execution state. A semantic function *Prolog* of this kind is called a *static* or *collecting* semantics [16, 58] since it characterizes a program by collecting information during its execution.

We regard C as a domain of sets of execution states. To extend the SLD example further, the input to *Prolog* would be a set of AND/OR trees which are empty apart from the root (the entry predicate of the program) annotated with a substitution.

C inherits the set inclusion ordering of powersets; we call this ordering $\subseteq_C$. The least element is the empty set of execution states and the greatest element is the set of all possible execution states.

The *concrete* domain C is abstracted to give the *abstract* domain A. This will be a homomorphic, but smaller, image of C. Since we are interested in inferring properties of execution states, an element in A will be a description of some execution states

[1] We usually drop the P argument.

[2] Without loss of generality we assume the program has a single entry predicate.

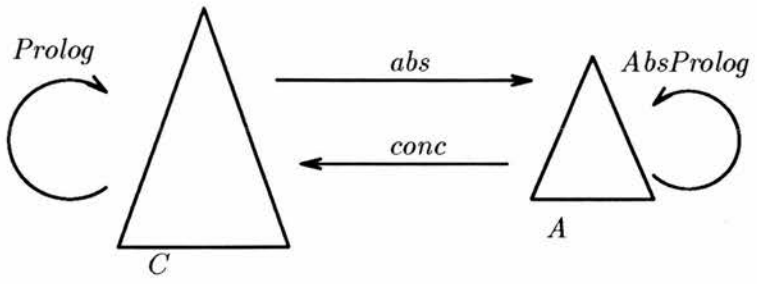


Figure 4.1: Relationship between C and A

in terms of those properties. Intuitively, such an element—which we call an abstract execution state—represents the set of all concrete execution states which have those properties. A also has an ordering, which is called $\subseteq_A$. An abstract execution state should be greater in $\subseteq_A$ than another if it represents a larger set of concrete execution states.

The homomorphism between the two domains is formalized by defining two mapping functions between them: abstraction $abs{:}C \rightarrow A$ and concretization $conc{:}A \rightarrow C$. These functions should satisfy the following:

$$\begin{aligned} &\forall c, d \in C,\ \forall a, b \in A \\ &\qquad c \subseteq_C d \quad \Rightarrow \quad abs(c) \subseteq_A abs(d) \\ &\qquad a \subseteq_A b \quad \Rightarrow \quad conc(a) \subseteq_C conc(b) \\ &\qquad\qquad c \quad \subseteq_C \quad conc(abs(c)) \\ &\qquad\qquad a \quad =_A \quad abs(conc(a)) \end{aligned}$$

The first two conditions state that *abs* and *conc* are monotonic.

To complete the abstract interpreter an abstract semantic function *AbsProlog* is needed. This will compute a result corresponding to the result of *Prolog* in the abstract domain $AbsProlog_P{:}A \rightarrow A$. The diagram (Figure 4.1) illustrates the relationship between C and A.

There are two important requirements that an abstract interpretation must meet to form the basis of automatic program analysers: correctness and termination.

4.2.1 Correctness

The information an abstract interpreter infers should always be correct[3], though as we have seen in Section 1.4 it may not always be precise. Correctness can be guaranteed by placing some conditions on the abstract interpretation.

Informally, the correctness condition states that if the abstract interpreter infers some description of the possible executions of a program and a set of queries then all

[3]Correctness is sometimes called soundness or safety

executions of the concrete interpreter with that program and set of queries should be included in that description; that is:

$$\forall c \in C, \quad Prolog(c) \subseteq_C conc(AbsProlog(abs(c)))$$

It is necessary to demonstrate that this holds for all programs and queries.

Note that if the abstract interpreter always returns some top element in $\subseteq_A$ which represents all possible execution states then the above condition will always hold. However, no useful properties can be inferred from this top element; that is, the result is completely correct but totally imprecise. One of the aims in designing an abstract domain is to ensure that it can satisfy the above condition and produce an output which is as small (with respect to $\subseteq_A$), and therefore as precise, as possible.

We will return to the issue of ensuring correctness in the following chapter (Chapter 5) where we develop our framework for abstract interpretation.

4.2.2 Termination

To be automatic, a program analyser based on abstract interpretation must guarantee that any execution will terminate. Semantics for recursive programs usually involve some iterative, often bottom-up, fixpoint characterization of the function over some partially ordered set. For automatic abstract interpretation it is necessary to place some restrictions on the abstract domain A so that this characterization can be finitely computed.

An obvious restriction is to make A finite; any fixpoint computation over a finite set is guaranteed to terminate. This is the solution most researchers have adopted but it is, in general, too strong a restriction.

Computations over infinite sets can be made to terminate provided that the set does not contain an indefinitely increasing chain (that is a sequence $a \subseteq_A b \subseteq_A c \ldots$ where the elements $a, b, c \ldots$ are distinct). An iterative bottom-up computation involves stepping along some chain. Removing all infinite chains will ensure that these computations terminate. This is the termination condition used by Bruynooghe [7] in his Prolog abstract interpretation framework.

There is an alternative way of viewing this restriction. Infinite chains may be permitted provided that in practice no iteration tries to go up them. That is, the subset of A used in a particular analysis should conform to this restriction, even if A does not. Consider an abstract domain characterizing lists as L_n which represents all lists with length n or less. Thus the infinite chain $L_0 \subseteq_A L_1 \subseteq_A L_2 \ldots$ occurs in A. An iteration along this chain may be terminated by adding a new token L, which represents all lists, into the abstract domain, provided the iteration can be made to 'jump' straight to this general description of lists. This approach gains some flexibility

in deciding at which point this generalization should occur[4].

We will return to the issue of ensuring termination in the following chapter (Chapter 5) where we develop our own framework for abstract interpretation.

4.2.3 Predicate Calls and Exits

The description of abstract interpretation given in the previous sections is very general; indeed it can apply to any language, not just Prolog. Here we recast some of the above to make it more appropriate to the kind of applications we have in mind.

For our applications, as well as many others (as we will show in Section 4.3), the entire execution state of the program is not needed.

Rather we are interested in characterizing the possible substitutions at points before and after every subgoal in every clause body in the program[5]. This means that the part of the 'execution state' we are interested in is a mapping from these points to sets of substitutions. An abstract execution state is a mapping from the same program points to *abstract* substitutions each of which describe a set of concrete substitutions.

Since both C and A will contain this program point mapping it is convenient to factor it out from the domains and incorporate it into the collecting semantics. These changes lift the type of the concrete collecting semantics to $Prolog_{PPts}{:}C{\rightarrow}(Pts{\rightarrow}C)$ and that of the abstract collecting semantics to $AbsProlog_{PPts}{:}A{\rightarrow}(Pts{\rightarrow}A)$, where Pts is the set of program points of interest. The correctness condition is restated:

$$\forall p \in Pts,\ \forall c \in C, \quad Prolog(c)(p) \ \subseteq_C \ conc(AbsProlog(abs(c))(p))$$

We are then left with C being the powerset of the set of all concrete substitutions. Note also that since we have abstract substitutions representing sets of concrete substitutions, we can naturally talk of abstract terms which represent sets of concrete terms.

4.2.4 The Groundness Example Returned

To illustrate the above ideas, we return to the groundness example of Section 1.4 which attempted to characterize the groundness of arguments in each goal. The program being analysed was:

```
p(f(A), B) :- q(A), C=g(A), r(C), D=g(B), s(D).
?- p(f(a), X).
```

The program points of interest are in between subgoals in the text of the program. We identify these using $\star_n$:

[4]This generalization mechanism is used in OLDT resolution to ensure completeness of Prolog [71].

[5]Often, we also need to have a specification of which variables are involved in the subgoals before and after the point so that it is possible to relate the substitutions to these subgoals. This information, however, is easily added afterwards.

p(f(A), B) :- $\star_1$ q(A), $\star_2$ C=g(A), $\star_3$ r(C), $\star_4$ D=g(B), $\star_5$ s(D) $\star_6$.
?- $\star_7$ p(f(a), X) $\star_8$.

The collecting semantics gathers a set of substitutions for each of these points, though these sets will be singleton since there is no recursion or backtracking in this program. For the concrete interpreter the result is:

$\star_1 \rightarrow \{ \langle$ A/a B/X C/C D/D $\rangle \}$
$\star_2 \rightarrow \{ \langle$ A/a B/X C/C D/D $\rangle \}$
$\star_3 \rightarrow \{ \langle$ A/a B/X C/g(a) D/D $\rangle \}$
$\star_4 \rightarrow \{ \langle$ A/a B/X C/g(a) D/D $\rangle \}$
$\star_5 \rightarrow \{ \langle$ A/a B/X C/g(a) D/g(X) $\rangle \}$
$\star_6 \rightarrow \{ \langle$ A/a B/X C/g(a) D/g(X) $\rangle \}$
$\star_7 \rightarrow \{ \langle$ X/X $\rangle \}$
$\star_8 \rightarrow \{ \langle$ X/X $\rangle \}$

Whereas the abstract interpreter returns the following:

$\star_1 \rightarrow \langle$ A/g B/a C/a D/a $\rangle$
$\star_2 \rightarrow \langle$ A/g B/a C/a D/a $\rangle$
$\star_3 \rightarrow \langle$ A/g B/a C/g D/a $\rangle$
$\star_4 \rightarrow \langle$ A/g B/a C/g D/a $\rangle$
$\star_5 \rightarrow \langle$ A/g B/a C/g D/a $\rangle$
$\star_6 \rightarrow \langle$ A/g B/a C/g D/a $\rangle$
$\star_7 \rightarrow \langle$ X/a $\rangle$
$\star_8 \rightarrow \langle$ X/a $\rangle$

Note that we have defined an abstract substitution to represent a set of concrete substitutions, so there are no explicit sets on the right hand side of this mapping.

This result is correct because, for each variable, each value in the concrete substitutions is included in the concretization of the value in the abstract substitution. That is the terms a and g(a) are members of the set represented by g and the variables and the terms X and g(X) are members of the set represented by a.

For this program the analysis terminates because there is no recursion or backtracking. In general, all analyses using this abstract domain can be made to terminate because the abstract domain is finite. This is because there can only be a finite number of variables in the arguments of a subgoal, and because the set of abstract values that these variables may take is also finite.

4.3 Example Applications

Abstract interpretation has recently been used in a wide variety of applications in logic programming. These include:

- Mode inference [14, 51, 22, 40].
- Type inference [8, 44, 55].
- Occur check reduction [68].
- Variable aliasing [79, 34, 57].
- Shared data-structures [9, 55]
- Program specialization [30, 28, 29].

We now take a closer look at some of the applications of abstract interpretation. Typically, abstract interpretation research has been driven by the desire to build better compilers[6] and the applications chosen here reflect that. However, they also form the existing applications most relevant to our techniques spotting application.

4.3.1 Mode Inference

A mode of a goal is the call time instantiation pattern of its arguments; that is whether they are free or instantiated. Because of the usefulness of mode information in optimizing Prolog compilers [50], mode inference is an important application of abstract interpretation. Abstract interpreters for mode inference attempt to characterize all the possible calls to each predicate in the program in terms of the instantiation state of their arguments. This is equivalent to characterizing the substitutions just before each goal in terms of the instantiation state of the variables.

Abstract domains for mode inference map variables onto a small set of tokens; Debray [22] uses tokens for certainly ground, certainly free, anything and no information. In an attempt to gain precision, Mellish [49] uses a slightly more complex domain which includes a token for instantiated terms which have free arguments.

Both of these abstract domains lose precision through not being able to represent aliasing between variables; that is variables which have been bound together. This leads to a considerable loss of information in some circumstances. For example, take the program:

[6] Not exclusively however—the information Lever [44] infers is used as documentation for the programmer.

```
p(X,Y) :- q(X,Y), r(X), s(Y).
q(X,X).
r(a).
s(_).
```

The variables X and Y in p/1 are aliased by the call to q/2. When the call to r/1 grounds X it also grounds Y, so the call to s/1 is also ground. Using an abstract interpreter based on the above abstract domains, any unification which could result in aliasing between variables must bind these variables to the anything token; this is the only way to ensure that the analysis will produce correct information. If the variables were left bound to the variable token the groundness of X would not be propagated to Y and the call to s/1 would appear to involve a free variable.

It is possible to add aliasing information to the abstract domain to avoid these problems; this is the approach used by Bruynooghe *et al* [9]. Alternatively, an abstract domain can simply not attempt to detect free variables. This is the approach used recently by Debray [19]. The abstract domain uses only the tokens: certainly ground, certainly instantiated and any[7]. This infers good quality information when aliasing is not an issue, as is the case in a large majority of everyday Prolog programs. It is also very simple and can be efficiently implemented. This abstract domain illustrates the trade-off between designing an abstract domain which infers the information needed and designing an abstract domain which can be efficiently implemented.

4.3.2 Call/Exit Type Inference

Even though Prolog is a single-typed language, it is natural to group certain sets of data-structures together into types. For example, lists form a natural type parameterized on the type of the elements contained. A 'call type' is some specification of the data-structures that can possibly appear in a subgoal's arguments at call time [54]. Since such arguments are not necessarily ground, a call type for a predicate subsumes the mode of that predicate. Analogously, the exit type of a predicate is some specification of the data-structures that can possibly appear in a subgoal's arguments at exit time.

This notion of type is in contrast to the conventional computer science notion of typing as exemplified in the functional language ML [33]. In terms of Prolog, this corresponds to a type of a predicate specifying some superset of the atoms which are true in the model of the program [60, 80]. Consider the standard Prolog predicate append/3:

```
append([], L, L).
append([H|L1], L2, [H|L3]) :-
   append(L1, L2, L3).
```

[7]Note that the groundness abstract domain we have used as an example (see Section 1.4) is a simplification of this domain.

Append/3 is used for joining or breaking up lists; that is, the type of each argument is a list. The call type of append/3 in some program might be:

append(*list(integer)*, *list(integer)*, *var*).

and the success type:

append(*list(integer)*, *list(integer)*, *list(integer)*).

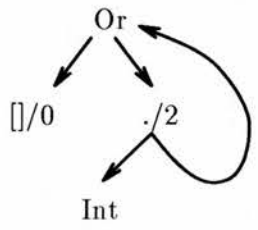
Here *list(integer)* is the name for the type of lists with integer elements. However, the model for append/3 includes atoms such as append([],a,a) which follow from the first clause. This atom is not in the call or success type specification but should appear in the 'conventional' typing for append/3.

Type inference for both notions of typing can be obtained by abstract interpretation. However, here we describe only call/success typing.

The two most significant pieces of recent work are the type inferencing schemes of Bansal [4] and Bruynooghe and Janssens [8]. They are both fairly similar in that an abstract substitution is a mapping from variables to type trees. These trees are the standard tree representation of terms supplemented with:

- nodes for representing a choice;
- backarcs for representing recursive data-structures.

For example, the 'list of integer' type above might appear as:

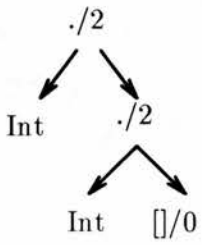


where 'Or' represents the choice node, 'Int' represents all integers.

The main difference between these two domains is the way in which aliasing between values is represented. Bruynooghe does not distinguish between variables in trees—that is all variables are represented by a special token—and sharing is represented by two separate components specifying certain and possible aliasing between variable nodes. For technical reasons, the domain cannot represent certain variable aliasing inside recursive data-structures.

Bansal, on the other hand, can distinguish between variables in trees, and so there is no need for the extra components. This has corresponding gains in the simplicity of the algorithms, and also their proofs, used to implement unification in his interpreter.

The abstract domain is infinite, but with backarcs and 'or' nodes an infinite set of increasing data-structures may be represented in one tree. The primitives in the abstract interpreter must be designed so that backarcs are introduced if a potentially unbounded data-structure appears to be being built. The primitives described by Bruynooghe look out for repeated constants on the branches of the tree being built. That is, if the tree appeared as:



then the repeated constant ./2 would be detected and the tree would be replaced by the 'list of integer' tree above.

Bansal does not deal with termination, assuming that the programs being analysed always terminate.

In Chapter 7 we sketch a type inferencing abstract domain. It is similar to the above schemes but allows the nodes representing choices to be linked. This enables certain relationships between data-structures to be specified, in addition to the data-structures' form. For example, it is possible to state that list *a* is the same length as list *b*. Our domain is also based on a different concrete description of Prolog.

4.3.3 Variable Aliasing

Considerable work has been done in trying to infer when subgoals in a clause execution are independent. Two subgoals are independent if they do not have any variables in common. A special case of this is when a subgoal is ground. If they are independent they may be executed in parallel with no communication between the separate processes [23]. For example, take the program:

```
contained(empty, _).
contained(node(LT, X, RT), L) :-
    member(X, L), contained(LT, L), contained(RT, L).
?- contained(T, L).
```

This determines whether all values appearing in the first argument, a binary tree, are members of the second argument, a list. The recursive calls in the second clause may execute in parallel if L is ground and LT and RT are independent. Because variables may be bound together, or aliased, during the program execution this cannot be inferred simply by checking the program text. With abstract interpretation it is possible to infer

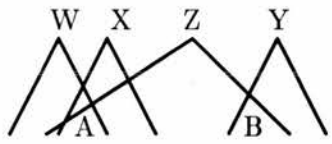


Figure 4.2: Example substitution where W, X, Y and Z are the local variables and A and B are the only non-local variables

at each point in the program some idea of which variables are certainly not aliased together and which are certainly ground. This information may be used to minimize the amount of independence checking needed at run time.

The most elegant abstract domain currently formulated for this application is that of Jacobs and Langen [34]. Their approach is to characterize each variable in the substitution with the set of local variables—that is the ones local to the currently executing clause—which contain it. The abstract substitution is the set of these sets. For example, the concrete substitution ⟨W/A X/A Y/B Z/f(A,B)⟩ where W, X, Y and Z are the local variables (see Figure 4.2) is represented as {{W,X,Z},{Y,Z}}. The substitution contains the set {W,X,Z} because the variable A is contained in W, X and Z, and it contains {Y,Z} because the variable B is contained in Y and Z. If there were a third non-local variable C contained only in Z the abstract substitution would be {{W,X,Z},{Y,Z},{Z}}.

This captures and propagates groundness and independence information with great precision. In the above, X and Y are independent because they do not appear in the same set, grounding either X or W grounds the other, and grounding Z grounds W, X, and Y. This can be inferred from the set representation.

Elegant though it is, however, even this domain suffers from the lack of other information which can seriously effect the quality of information inferred. For example, the lack of type information may cause some clause to be considered which would not in fact be executed in the concrete program. This is a problem with many abstract domains.

Similar domains have been used in the occur check reduction application. An essential part of the unification algorithm is the occur check which ensures that whenever a variable is about to be bound to some term then that term does not contain that variable. If it does then the unification should fail. The overhead of performing this check is so great that if it were implemented it would seriously effect the performance of Prolog systems. Accordingly, the check is usually left out completely and circular bindings may occur. In this sense many Prolog implementations are unsound. A solution to this problem is to detect all unifications that cannot possibly cause a circularity.

This is the purpose of the work of Plaisted [62] and Søndergaard [68].

4.3.4 Detecting Shared Data-Structures

Another way of improving code generated by a Prolog compiler is to ensure that it is capable of reusing space that can be shown at compile time to be redundant; this is known as compile time garbage collection. Data-structures may then be reused during the computation, avoiding the cost of creating new space and effectively giving Prolog programs the efficiency of destructive assignment.

In a deterministic program, a data-structure which is bound to a variable may be reused if that variable is not used later on in the clause, provided that the data-structure does not share with any other data-structures which are used later on. Sharing is very much an implementational notion, treating the pointers that the Prolog system builds as a graph; two variables share if their graphs contain a common subgraph. A related notion is containment; a variable contains another if its graph completely contains the other variable's graph. Thus the information required is a partitioning of the local variables into sets between which there is no sharing at points just before each subgoal in a clause. This can be seen as a data-structure equivalent of variable aliasing.

Bruynooghe *et al* [9] have sketched an abstract domain where an abstract substitution consists of two sets of pairs of variables[8]. One of these sets represents the pairs of variables X, Y which possibly share, each of which are written as X AL Y. The other represents the pairs of variables of which the first one is certainly contained by the other, which is written as X PART Y. Note that the AL relation is reflexive and symmetric, but the PART relation is only reflexive. Also, if X PART Y then X AL Y. We write an abstract substitution as a set of these pairs, but the extra pairs implied by these reflexive and symmetric properties are not given explicitly.

For example, consider executing the following query with an initial substitution where all variables are known not to share, that is {}:

```
?- X = f(A,A), X = f(B,C).
```

After execution of the first unification, the substitution is:

{ A PART X }

since A's graph is certainly contained in the graph of X. After the second unification the substitution is:

{ X AL B, X AL C, A AL B, A AL C, B AL C, A PART X, B PART X}

[8]It contains somewhat more information which is specific to their application but which we ignore at present.

that is all the variables possibly share with each other. Note that because the actual constants used to make up a variable's graph are not represented in the abstract substitution it is impossible for abstract unification to determine where a variable appears in a graph. This partly leads to the poor quality information derived above.

Another cause of this is that sharing inside a variable's value cannot be represented because the same variable cannot appear on both sides of a pair. This means that a data-structure which is a graph cannot be distinguished from one that is a tree. This forces a worst case assumption about sharing in unifications which leads to a considerable loss of precision. For example, take the call:

```
?- X=[H|T]
```

If X is known to be a tree then H and T do not share afterwards. If X is a graph, H and T will in general share.

This representation problem is solved by Mulkers *et al* [55] by merging a type inferencing component with the possible sharing set. Here the set is composed of pairs of nodes in the type trees rather than pairs of variables. This abstract domain is essentially a merge of the type inferencing domain of Bruynooghe and the simple data-structure sharing domain above.

Recently, Winsborough [78] has developed another simple domain for the garbage collection application. The domain is based on an instrumented concrete description of Prolog[9] which uses substitutions augmented by a set of pairs of paths indicating positions inside terms that share their representation. An abstract substitution consists of a triple of three sets: a set of variables which are certainly ground, a set of pairs of variables which may be bound to terms that have a variable in common and a set of pairs of variables which may be bound to terms which share. An algorithm for abstract unification is described over these substitutions.

The abstract domain we develop in Chapter 6 derives similar information to these abstract domains but is more precise than Bruynooghe's and simpler than Mulkers'. Also it is based on a different instrumented concrete description of Prolog, so it is impossible to simply take Bruynooghe's nor Mulkers' abstract domains without any adaptation. Winsborough's domain infers sharing relations only, whereas we require inclusion relations as well.

4.3.5 Program Specialization

Program specialization involves constructing a new program which can follow only certain computational paths of the original one, but not others. The work of Gallagher *et al* [30, 28] has developed techniques for specializing both Prolog and Flat Concurrent

[9] 'instrumented' means that it has been supplemented with extra features needed for the abstract domain but not actually required by Prolog.

Prolog programs [66]. Abstract interpretation plays a part by its ability to characterize all possible computational paths which a program may follow. This characterization is of course approximate but any parts of the original program which contribute nothing to this characterization may be safely removed in the new program.

An elegant example of a program specialization scheme is given by Gallagher and Bruynooghe [29]. They describe an abstract interpretation scheme which characterizes the most general call made to each predicate in a program. With this information it is possible to remove redundant constructor functions. For example, in the standard flatten predicate using a constructor function -/2 to join together the two difference list arguments:

```
flatten(L, FL) :- flatten(L, FL-[]).

flatten([H|T], L-L0) :-
    flatten(H, L-L1),
    flatten(T, L1-L0).
flatten([], L-L),
flatten(X, [X|L]-L) :-
    X\=[], X\=[_|_].
```

The most general call to flatten/2 is flatten(A,B-C); that is the second argument of every call to flatten/2 contains the -/2 constructor function. Knowing this it is possible to specialize flatten/2 by replacing all occurrences of -/2 with a pair of separate arguments resulting in considerably faster code:

```
flatten(L, FL) :- flatten(L, FL, []).

flatten([H|T], L, L0) :-
    flatten(H, L, L1),
    flatten(T, L1, L0).
flatten([], L, L),
flatten(X, [X|L], L) :-
    X\=[], X\=[_|_].
```

We assume that this transformation has been performed on all our student example Prolog code (see Section 1.6).

4.3.6 Conclusions

There is not much difference between most of the applications described in this section; they typically attempt to characterize the substitution at various points in a program's execution. Exactly which points they use is dependent on the application; for example, type inference may need the exits of subgoals as well as calls, and for specialization it may be necessary to distinguish between calls to textually different subgoals. Note,

that a simple generalization of the points used in all these applications is the points before and after every subgoal in the text of the program. It is straightforward to map from this information to a more appropriate form for each application.

Most of the abstract domains used are fairly simple. Generally, the smaller the abstract domain the more efficient it is to implement. When it is possible to design a small abstract domain which precisely infers information for many everyday programs, abstract interpretation undeniably forms the basis of practical program analysers.

In Chapter 6 we describe a simple abstract domain for inferring inclusion and sharing relationships. It derives similar information to the abstract domains for detecting shared data-structures (Section 4.3.4), but is based on a different concrete description of Prolog so it is impossible to make use of these abstract domains without some adaptation.

Unfortunately, a small, elegant domain can sometimes produce imprecise information because it contains too few details of the concrete Prolog execution. In such cases it is tempting to add these details into the abstract domain, but this can lead to domains which are too complicated to demonstrate correctness or to implement efficiently.

There is a case for developing a 'large' abstract domain which infers information which subsumes all the current applications. This general analysis would be performed once on each program. If an abstract domain can be formulated which successfully reaches a compromise between the different applications and which can be implemented, this may be preferable, especially since most current applications are for driving optimizing compilers and ignore the usefulness of having information about programs available during program development and debugging. We sketch a design for such a large abstract domain in Chapter 7. This is based on call/exit type inferencing abstract domains (Section 4.3.2) but allows the nodes representing choices to be linked. This enables certain relationships between data-structures to be specified, in addition to their form. For example, it is possible to state that list a is the same length as list b. It is also based on a different concrete description of Prolog.

4.4 Types of Collecting Semantics

Here we describe some of the Prolog collecting semantics which have been used as the basis of abstract interpreters. Such collecting semantics form the basis of frameworks for abstract interpretation.

A suitable collecting semantics must make explicit the substitution just before and after each call and should reflect the standard Prolog execution. A collecting semantics should also make it as easy as possible to prove the correctness of any abstract interpreter based on it. Since all Prolog semantics contain a similar set of primitives—such as unification—this is in practice more dependent on the abstract domain used.

Another important consideration when selecting a suitable collecting semantics is to ensure that any abstract interpreter based on it will always terminate. Semantics are usually regarded as specifications and so do not take termination into account. As we have seen above (Section 4.2.2), the main condition for ensuring termination is to restrict the abstract domain to be, in some sense, finite. This assumes that the semantics characterizes recursive Prolog predicates by using an iterative bottom-up component. The simplest description of Prolog execution—the vanilla interpreter—has no such iterative component. This interpreter can enter non-terminating loops even when the domain over which it is executing is finite, and so it is not possible to take this as the basis for an abstract interpreter without some modification.

Following the classification given by Kanamori [38], there are three approaches to solving this problem, each leading to a different style of abstract interpreter: pure bottom-up abstract interpretation, one-phase hybrid abstract interpretation and two-phase hybrid abstract interpretation. See also Mellish [52] for an account of constructing several different abstract interpreters by using specialization of combinations of interpreters and so highlighting the similarities between the three approaches.

4.4.1 Pure Bottom-Up Abstract Interpretation

The first approach is to base a collecting semantics on a fixpoint characterization of a bottom-up semantics, such as the T_p operator of van Emden and Kowalski [74, 45] or Fitting's Φ_p operator [27]. This approach is used in the type inference application of Kanamori and Horiuchi [39] and in the abstract interpretation framework given by Marriott and Søndergaard [47].

There are some features of this approach which make it unsuitable for our typical applications:

- These bottom-up semantics are not *query directed* [48]; that is they do not limit their characterization of the program to only what should occur for a given query and so potentially do more work than is necessary and give less precise results.
- Bottom-up semantics such as T_p only describe ground instances of atoms.

However, they are suitable for other applications. For example, it is possible to formulate 'conventional' type inference (such as that of Mycroft and O'Keefe [59], see Section 4.3.2) as a bottom-up abstract interpretation.

Bottom-up collecting semantics can also be suitable for the abstract interpretation of languages whose natural execution is bottom-up, such as Datalog [31].

There have recently been attempts at overcoming some of the above problems. The Prolog semantics of Falaschi *et al* [26] is bottom-up but characterizes a program in terms of non-ground atoms. The abstract interpretation framework of Kemp and Ringwood [42] is based on this semantics.

4.4.2 One-Phase Hybrid Abstract Interpretation

One-phase hybrid abstract interpretations are based on a semantics which models the standard Prolog execution (providing the information needed in our applications) supplemented with a bottom-up mechanism which is used whenever encountering a recursive call which might lead to non-termination.

This mechanism can be seen as an extension of memoization [53] called *extension tables*. Dietrich [24] develops a Prolog interpreter which incorporates an algorithm *ET** for building extension tables. The idea is to globally store in a table each call to a predicate together with all its exits. If another recursive call is made which is subsumed by—that is, is at least as specific as—an earlier one then this call returns the set of exits (with variables suitably renamed) in the table entry of that call and then is immediately failed rather than executed further. When the original call has finished execution, the entire computation is repeated until no new exits are added into the table, at which point the execution stops. The naive implementation of *ET** can be quite inefficient.

A mode inference abstract interpretation scheme based on this idea has been given by Debray [21, 22]. The implementation is made more efficient by a variety of schemes one of which is using specialization to remove the overhead of the interpreters.

Apart from efficiency, a more fundamental criticism of this approach is that the bottom-up mechanism is *ad hoc* in that it has no correspondence to any standard semantics for Prolog [45]. An improvement is the abstract interpretation framework developed by Bruynooghe [7]. This defines a procedure for constructing an abstract AND/OR tree for a program which is shown to be correct by providing a mapping from the tree to an (infinite) set of AND trees that correspond to all SLD resolutions possible from that program. Potentially nonterminating recursive calls are handled by providing for a fixpoint computation when repeated calls to the same predicate are detected. A similar scheme is described by Corsini and Filè [15].

An altogether more elegant approach is to incorporate a bottom-up mechanism directly into a proof theory for Prolog, and then demonstrate the soundness and completeness of this theory; this is done in Tamaki and Sato's OLDT resolution [71]. The bottom-up mechanism is essentially the same as *ET** but OLDT also allows calls to be generalized in certain situations so that it is possible for completeness to be guaranteed even for languages such as Prolog with function symbols which allow 'increasing' non-terminating loops. For example, take the program:

```
p(X) :- p(f(X)).
?- p(a).
```

This can be made to terminate by generalizing the original call p(a) to p(X) which is more general than the first recursive call p(f(a)).

The naive implementation of this semantics is essentially the same as the *ET**

algorithm, though it is possible to define a more elegant interpreter based on a suspension mechanism [64]. An interpreter implementing OLDT resolution can be readily abstracted; see for example the work of Kanamori *et al* [40, 41].

Note also that the extension table completed at the end of the computation forms a suitable output for the collecting semantics our typical applications above require, and so there is often no need to add any further complication to the interpreter to construct the collecting semantics output. Because of this and the above reasons we base our own framework for abstract interpretation (developed in Chapter 5) on the OLDT approach.

Another approach to one-phase hybrid abstract interpretation is to take a fixpoint denotational semantics for Prolog which captures Prolog's standard execution and add to it a mechanism to ensure that it is query directed. Such a denotational semantics is given by Jones and Søndergaard [37]. A suitable mechanism is a *minimal function graph* introduced for applicative languages by Jones and Mycroft [36] and applied to logic programming by Winsborough [77] and Gallagher and Bruynooghe [28]. A minimal function graph represents the part of a program's total meaning relevant to a particular query. Although this mechanism appears to "bridge the gap between program analysis frameworks based on operational semantics and those based on denotational semantics" [28], it does so by introducing considerably more machinery into the semantics. It is also interesting to note that the implementation of the scheme Gallagher and Bruynooghe describe is essentially the same as OLDT resolution, so in practical terms little has been achieved.

4.4.3 Two-Phase Hybrid Abstract Interpretation

Here a top-down semantics, usually combined with the program to be analysed, is transformed into a form which can be executed by a bottom-up procedure and still capture the top-down characteristics of the program execution. The analysis is broken into two phases: the transformation and then conventional bottom-up execution. Thus the top-down execution strategy of Prolog is modelled by a bottom-up iterative computation.

This approach was first taken by Mellish [51] for mode inference. Starting with operational Prolog semantics which characterizes a program as a set of traces, he then presents mutually recursive functions generating the set of all the call and exit atoms which is shown to be correct with respect to the traces. These functions are abstracted in the usual way; that is, by replacing the primitives over the concrete domain with corresponding primitives over the abstract domain. Specializing these functions using the object program produces a set of simultaneous equations which are then solved bottom-up.

Two-phase hybrid abstract interpretation has the advantage that the bottom-up

procedure can sometimes be executed efficiently (for example, see O'Keefe [61]).

Recently, the similarity between this approach and the Magic Sets type transformations originally used in deductive databases for evaluating queries efficiently [63, 3, 65] has been noticed. These transformations also take a program and transform it into a new program which when executed bottom-up accurately models the top-down execution of the original program. Abstract interpretation frameworks developed from this approach are given by Debray and Ramakrishnan [20], Kanamori [38], and Codish *et al* [13]. It can be shown that OLDT and the Magic sets approach are equivalent; see Bry [10].

4.4.4 Improving Efficiency

We now describe some approaches that attempt to avoid altogether the overhead of the bottom-up component. These are not in general applicable, but simply take advantage of certain characteristics of the particular abstract domain being used.

- Lever [44] characterizes certain forms of recursive predicates which compute their fix-point after one iteration.
- Codish *et al* [14] compute their fix-point from the top of the domain rather than the bottom. Since they are always above the least fix-point and hence always correct, it is possible to stop the iteration at any time and still yield sound information. In their implementation of a mode inferencing scheme for Flat Concurrent Prolog [72] they stop after just one iteration and because of the flatness of their domain, good quality, precise information is still inferred for many programs.
- Jacobs and Langen [34] first perform a bottom-up pre-analysis to produce a fixed approximation to the meaning of each clause. The main top-down analysis combines together the clause approximations appropriate to a particular query. This approach does not involve unacceptable loss of information because their abstract unification shares some commutativity like properties with concrete unification, and because their abstract domain is small.

Recently, Debray [19] has attempted to characterize the abstract domains which can always be implemented efficiently, though here he is interested in avoiding the overheads of having to maintain variable aliasing rather than avoiding iterative computations.

4.4.5 Conclusions

We have seen that one approach to guaranteeing termination is to incorporate some kind of bottom-up component in the abstract interpreter, assuming some finiteness

property of the abstract domain. In the abstract interpretation framework developed in the following chapter we incorporate an extension table mechanism into our interpreter shell. The resulting interpreter can be seen as an implementation of OLDT resolution. We choose this approach over the others discussed above because:

- Despite the complexity introduced by the extension table mechanism, we believe it is overall the simplest approach to Prolog abstract interpretation.
- A simple implementation is available in the work of Wærn [64].

4.5 Summary

In this chapter, we have

- introduced the ideas behind abstract interpretation and given some idea of what is involved in building a program analyser based on abstract interpretation.
- reviewed some recent 'typical' applications of abstract interpretation in logic programming.
- discussed some types of collecting semantics that have been used as the basis for abstract interpreters, and motivated our design for an abstract interpretation framework.

Chapter 5

A Framework for Prolog Abstract Interpretation

5.1 Introduction

In this chapter, we develop a *framework* for Prolog abstract interpretation to be used as the basis for the program analysers we describe in Chapters 6 and 7. This chapter can be seen as factoring out the common parts of these analysers. The rationale for providing such a framework is sometimes given as an attempt at reducing the proof effort needed to develop program analysers, but here the framework is used purely as a way of structuring our description. Accordingly, we make no claims as to the generality of the framework, even though it compares well to other frameworks [7, 47, 28].

For our techniques detecting application we require some characterization of the substitutions at each point before and after every subgoal in the text of the program. The information subsumes that needed for most of the applications we described in Chapter 4. The framework we describe here allows us to infer this information.

Our framework provides an interpreter shell which can be instantiated by an abstract domain to yield some particular program analyser. It is formulated so that, provided the abstract domain satisfies certain properties, the resulting analyser always yields correct information and always terminates. The interpreter we use implements OLDT resolution [71]. It is based on that of Wærn [64], but incorporates some improvements. The solution, or extension, table mechanism used in OLDT resolution adds considerably to the complexity of the interpreter. We therefore start by taking a simpler interpreter and introduce the full interpreter only when we come to ensuring termination.

```
prolog(true).
prolog((Goal_L,Goal_R)) :-
    prolog(Goal_L),
    prolog(Goal_R).
prolog(Goal) :-
    clause(Goal, Body),
    prolog(Body).
```

Figure 5.1: The 'vanilla' Prolog interpreter

5.2 An Interpreter Shell

Here we begin to develop interpreters which implement the $Prolog_{PPts}{:}C{\rightarrow}(Pts{\rightarrow}C)$ and $AbsProlog_{PPts}{:}A{\rightarrow}(Pts{\rightarrow}A)$ functions introduced in Chapter 4. For the moment, however, we ignore the program points *Pts*. We add them later in Section 5.3 when we consider correctness.

Prolog's execution is often described in terms of an interpreter written in Prolog itself; this forms the basis for many novel uses for Prolog. The usual implementation used is the 'vanilla' interpreter given in Figure 5.1. We will stick to using Prolog as the implementation language for our framework since this is both natural and convenient. However, the 'vanilla' interpreter is not suitable for the following reasons:

- It takes a single substitution as input not a set of substitutions as is required for implementing the collecting semantics *Prolog* introduced in Chapter 4.
- It non-deterministically returns individual substitutions as output rather than a single set of substitutions.
- It does not make substitutions explicit so we cannot easily convert it into a collecting interpreter which gathers substitutions at various program points.

We solve the above problems by adopting the somewhat more complex interpreter given in Figures 5.2 and 5.3. This interpreter takes a single query and a set of substitutions as input. We need not explicitly allow for a set of queries as input since it is possible, without loss of generality, to add a new predicate (say 'main') to the program which simply calls each element in the set of queries. The top level call to prolog/3 would then be '?- prolog(main(...), Input, Output).' which corresponds to the function *Prolog*. We assume that the program input to the interpreter is available through the predicate get_clauses/2; strictly it should be passed around explicitly as an extra argument.

The interpreter manipulates substitutions explicitly and thus includes calls to the primitives over substitutions hidden in the 'vanilla' interpreter. Note that join/3 is a new primitive required for joining together the sets of substitutions produced by

```
prolog(true, Set, Set).
prolog(L=R, Set_0, Set) :-
    bodyunify(L, R, Set_0, Set).
prolog((Goal_L,Goal_R), Set_0, Set) :-
    prolog(Goal_L, Set_0, Set_1),
    prolog(Goal_R, Set_1, Set).
prolog(Goal, Set_0, Set) :-
    get_clauses(Goal, Clauses),
    trim(Set_0, Goal, Set_Goal),
    prolog_clauses(Clauses, Goal, Set_Goal, Set_Clauses),
    compose(Set_0, Set_Clauses, Set).

prolog_clauses([], _, _, {}).
prolog_clauses([(Head:-Body)|Clauses], Goal, Set_0, Set) :-
    headunify(Head, Goal, Body, Set_0, Set_1),
    prolog(Body, Set_1, Set_2),
    restrict(Set_2, Head, Goal, Set_Body),
    prolog_clauses(Clauses, Goal, Set_0, Set_Clauses),
    join(Set_Body, Set_Clauses, Set).
```

Figure 5.2: A set-to-set Prolog interpreter

```
get_clauses(Goal, Clauses) :-
    Clauses is a list of all clauses which may match Goal.

trim(Set, Goal, Set_Goal) :-
    Set_Goal is Set with variables not accessible from Goal removed.

compose(Set_Orig, Set_Clauses, Set) :-
    Set is Set_Orig updated with the new bindings in Set_Clauses.

restrict(Set_Body, Head, Goal, Set_Goal) :-
    Set_Goal is Set_Body with variables not accessible from Head removed,
    and then any left renamed to the corresponding variable in Goal.

bodyunify(Left, Right, Set_Before, Set) :-
    Set is the set of successful unifications of
    Left and Right with each member of Set_Before.

headunify(Head, Goal, Body, Set_Call, Set) :-
    Set is the set of initial substitutions for executing Body.

join(Set_Body, Set_Clauses, Set) :-
    Set is the union of Set_Body and Set_Clauses.
```

Figure 5.3: Subsidiary predicates for the set-to-set Prolog interpreter

different clauses. The interpreter can only be regarded as a specification because in general the sets constructed may be infinite.

Because of the normal form transformation (see Section 1.6) we assume for Prolog code being analysed, we need the interpreter to allow for explicit body unifications (calls to =/2). This is handled as a special clause for prolog/3.

The interpreter returns a set of substitutions and implements a function rather than a relation, though we retain the usual Prolog notation. Since it operates over sets, the interpreter also loses the sequencing of Prolog's answers. However, for our applications this order does not matter. In any case, it can also be quite difficult to infer much about the order of solutions with an abstraction because it is only possible to approximate Prolog's head matching process.

As described in Chapter 4, an abstract interpreter is a concrete interpreter with the primitives operating over the concrete domain of sets of concrete substitutions replaced by primitives operating over the abstract domain of single abstract substitutions. There is therefore a correspondence between the types of the concrete and abstract interpreters. The shell for an abstract interpreter given in Figure 5.4 is similar to the concrete interpreter. Apart from the change of the variable names from 'Set' to 'A', the only change is the use of $\perp$ instead of $\{\}$ as a generalization of the null result. We assume that $\perp$ is the bottom element in the $\subseteq_A$ ordering on any abstract domain A.

It is possible to add arguments to the program; either to the entire interpreter or to individual primitives. For example, arguments may be added for ensuring variables are properly renamed when using a ground representation of object variables. The above abstract interpreter is the minimum necessary to correspond to the concrete interpreter.

In Section 4.2.2 we introduced the notion of generalizing an abstract substitution so as to ensure termination or to terminate quicker. It is possible to insert a predicate call which generalizes the current substitution at any point in the abstract interpreter.

5.3 Correctness

For results to be correct, an abstract interpreter must be consistent with the concrete interpreter. The correctness condition (first given in Section 4.2.1) can be stated:

$$\forall p \in Pts,\ \forall c \in C, \quad Prolog(c)(p) \ \subseteq_C \ conc(AbsProlog(abs(c))(p))$$

Note that this treats *Prolog* as a function, rather than a relation. The interpreter above implements a function, although it is written in a relational notation. To illustrate we give the abstract interpreter in a functional notation in Figure 5.5. Also the primitives are functions as well. In this section we use a functional notation throughout and

```
abs_prolog(true, A, A).
abs_prolog(L=R, A_0, A) :-
   bodyunify(L, R, A_0, A).
abs_prolog((Goal_L,Goal_R), A_0, A) :-
   abs_prolog(Goal_L, A_0, A_1),
   abs_prolog(Goal_R, A_1, A).
abs_prolog(Goal, A_0, A) :-
   get_clauses(Goal, Clauses),
   trim(A_0, Goal, A_Goal),
   abs_prolog_clauses(Clauses, Goal, A_Goal, A_Clauses),
   compose(A_0, A_Clauses, A).

abs_prolog_clauses([], _, _, ⊥).
abs_prolog_clauses([(Head:-Body)|Clauses], Goal, A_0, A) :-
   headunify(Head, Goal, Body, A_0, A_1),
   abs_prolog(Body, A_1, A_2),
   restrict(A_2, Head, Goal, A_Body),
   abs_prolog_clauses(Clauses, Goal, A_0, A_Clauses),
   join(A_Body, A_Clauses, A).
```

Figure 5.4: An abstract Prolog interpreter

assume a correspondence between the predicate and functional form for the primitives and the interpreter as a whole.

This condition may be quite difficult to prove. Here we justify some conditions over the primitives implementing the concrete and abstract domains which imply this correctness condition, and which are more convenient to prove. The conditions are:

$$\forall h, g, i \in \mathit{Terms},\ \forall a, b \in A$$

$$\begin{aligned}
\mathit{AbsTrim}(a,h,b) &\supseteq_A \mathit{abs}(\mathit{Trim}(\mathit{conc}(a),h)) \\
\mathit{AbsCompose}(a,b) &\supseteq_A \mathit{abs}(\mathit{Compose}(\mathit{conc}(a),\mathit{conc}(b))) \\
\mathit{AbsHeadUnify}(h,g,i,a) &\supseteq_A \mathit{abs}(\mathit{HeadUnify}(h,g,i,\mathit{conc}(a))) \\
\mathit{AbsBodyUnify}(h,g,a) &\supseteq_A \mathit{abs}(\mathit{BodyUnify}(h,g,\mathit{conc}(a))) \\
\mathit{AbsRestrict}(a,h,g) &\supseteq_A \mathit{abs}(\mathit{Restrict}(\mathit{conc}(a),h,g)) \\
\mathit{AbsJoin}(a,b) &\supseteq_A \mathit{abs}(\mathit{Join}(\mathit{conc}(a),\mathit{conc}(b)))
\end{aligned}$$

where *Terms* is the set of Prolog terms and each function corresponds to a predicate in the concrete or abstract interpreters.

The first step in justifying these conditions is to view an execution of both the abstract and concrete interpreter for some program as two trees where each node in the tree is a call to one of the primitives. The trees represent the composition of the primitive functions over sets of substitutions ignoring those parts of the program's execution which are the same on both the abstract and concrete interpreters. An

```
abs_prolog(true, A) =
   A.
abs_prolog((L=R), A) =
   bodyunify(L, R, A).
abs_prolog((Goal_L,Goal_R), A) =
   abs_prolog(Goal_R, abs_prolog(Goal_L, A)).
abs_prolog(Goal, A) =
   compose(A,
      abs_prolog_clauses(get_clauses(Goal), Goal, trim(A, Goal))).

abs_prolog_clauses([], Goal, A) =
   ⊥.
abs_prolog_clauses([(Head:-Body)|Clauses], Goal, A) =
   join(restrict(abs_prolog(Body, headunify(Head, Goal, Body, A)),
                 Head, Goal),
        abs_prolog_clauses(Clauses, Goal, A)).
```

Figure 5.5: The abstract Prolog interpreter written in a functional notation.

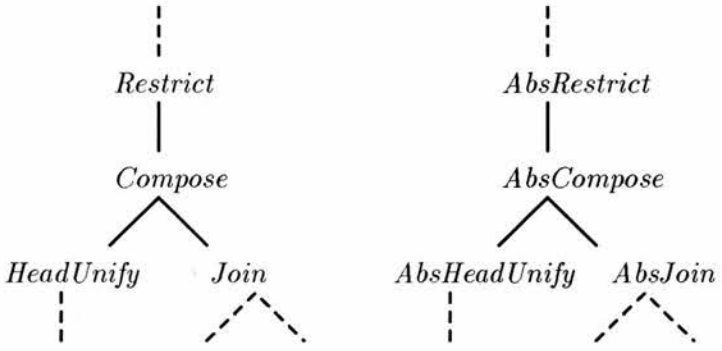


Figure 5.6: Corresponding fragments of trees representing a concrete and abstract execution of a program.

example of a fragment of two trees is shown in Figure 5.6. The abstract tree may contain branches which do not appear in the concrete tree, because of the approximate nature of abstract unification, but otherwise the two trees are isomorphic.

Each of the above conditions states that, for its particular primitive, the abstract version is a correct approximation of the concrete version. To be able to make statements about the interpreters as a whole we need to prove a similar statement for compositions of these primitives—that is, trees. For this, we can use an inductive argument over structure of the trees. The base case, where there is one primitive, is implied immediately by the corresponding condition above.

The inductive case assumes that the condition is true for a composition of n prim-

itives and shows that this implies it for a composition of $n+1$ primitives:

$$\begin{aligned}
&\forall a \in A, \quad AbsF_n(a) \supseteq_A abs(F_n(conc(a))) \\
&\qquad conc \text{ is monotonic} \\
&\Rightarrow conc(AbsF_n(a)) \supseteq_C conc(abs(F_n(conc(a)))) \\
&\text{since } c \subseteq_C conc(abs(c)) \\
&\Rightarrow conc(AbsF_n(a)) \supseteq_C F_n(conc(a)) \\
&\qquad \text{F is monotonic} \\
&\Rightarrow F(conc(AbsF_n(a))) \supseteq_C F(F_n(conc(a))) \\
&\qquad abs \text{ is monotonic} \\
&\Rightarrow abs(F(conc(AbsF_n(a)))) \supseteq_A abs(F(F_n(conc(a)))) \\
&\qquad \text{by assumptions} \\
&\Rightarrow AbsF(AbsF_n(a)) \supseteq_A abs(F(F_n(conc(a))))
\end{aligned}$$

Here F is any one of the above five primitives and $AbsF$ is its abstract equivalent. F_n is n (possibly different) primitives composed together and $AbsF_n$ is its abstract equivalent. The above proof assumes that each F is monotonic, the restrictions over the functions *abs* and *conc* (see Chapter 4) and the primitive correctness conditions given above.

We have simplified this proof by assuming each primitive takes only one substitution argument. To be more strict, it is necessary to rework the inductive step for different types of F(in particular, for *Compose* and *Join* which take two substitution arguments).

Since the entire interpreters are compositions of the primitives, our six conditions on the primitives allow us to make a similar statement for the output of the abstract interpreter:

$$\forall a \in A, \quad AbsProlog(a) \supseteq_A abs(Prolog(conc(a)))$$

The above inductive argument demonstrates that the set of concrete substitutions at every node in the concrete tree is correctly approximated by the abstract substitution in the corresponding node in the abstract tree. This allows us to infer the correctness statement over the program point mapping. A program point corresponds to a set of nodes in the trees so as long as the recording of information at each point is simply collecting the substitutions computed at the corresponding tree nodes. We can state the correctness of the program point mapping:

$$\forall p \in Pts, \forall a \in A, \quad AbsProlog(a)(p) \supseteq_A abs(Prolog(conc(a))(p))$$

Note that a point must lie between primitives in the interpreter, and not within a primitive. Thus, for example, we cannot collect substitutions inside unification.

It is straightforward to show that this implies the correctness condition we stated at the beginning of this section:

$$\begin{aligned}
\forall p \in Pts, \forall a \in A, \quad & AbsProlog(a)(p) && \supseteq_A && abs(Prolog(conc(a))(p)) \\
& \text{introduce } abs(x) = a \\
\Rightarrow \quad & AbsProlog(abs(x))(p) && \supseteq_A && abs(Prolog(conc(abs(x)))(p)) \\
& \text{by monotonicity and } c \subseteq_C conc(abs(c)) \\
\Rightarrow \quad & AbsProlog(abs(x))(p) && \supseteq_A && abs(Prolog(x)(p)) \\
& conc \text{ is monotonic} \\
\Rightarrow \quad & conc(AbsProlog(abs(x))(p)) && \supseteq_C && conc(abs(Prolog(x)(p))) \\
& \text{since } c \subseteq_C conc(abs(c)) \\
\Rightarrow \quad & conc(AbsProlog(abs(x))(p)) && \supseteq_C && Prolog(x)(p)
\end{aligned}$$

This proof uses the restrictions we have assumed for the functions *abs* and *conc* (see Chapter 4) and that the functions *Prolog* and *AbsProlog* are monotonic.

5.3.1 Variable Renaming

One problem we have ignored in the above is that of handling variable renaming. For us to compare two sets of substitutions using $\subseteq_C$ we need to ensure that the sets range over the same variables. Given that one of these sets has been through an abstraction/concretization process it is possible that the names of the variables have been lost.

To avoid this we assume we have a function *canon* which can take a set of substitutions and returns a set of substitutions in a canonical form. The correctness condition can then be restated:

$$\forall p \in Pts,\ \forall c \in C, \quad canon(Prolog(c)(p)) \ \subseteq_C \ canon(conc(AbsProlog(abs(c))(p)))$$

An alternative, which for simplicity we assume, is to incorporate the canonicalization in the concrete ordering; that is define

$$\forall c, d \in C, \quad c \subseteq_C d \ \equiv \ canon(c) \subseteq_{canon\,C} canon(d)$$

and similarly for the abstract ordering $\subseteq_A$.

5.3.2 Generalization

Lastly, to ensure correctness of any predicate implementing some generalizing function (see Section 4.2.2) *general* must satisfy:

$$\forall a \in A, \quad a \subseteq_A general(a)$$

```
abs_prolog(Goal, A, C) :-
    abs_prolog(Goal, A, _, C).

abs_prolog(true, A, A, C).
abs_prolog(L=R, A_0, A, C) :-
    bodyunify(L, R, A_0, A).
abs_prolog((Goal_L,Goal_R), A_0, A, C) :-
    abs_prolog(Goal_L, A_0, A_1, C),
    abs_prolog(Goal_R, A_1, A, C).
abs_prolog(Goal, A_0, A, C) :-
    get_clauses(Goal, Clauses),
    trim(A_0, Goal, A_Goal),
    abs_prolog_clauses(Clauses, Goal, A_Goal, A_Clauses, C),
    compose(A_0, A_Clauses, A),
    update(Goal, C, A_Goal, A).

abs_prolog_clauses([], _, _, ⊥, C).
abs_prolog_clauses([(Head:-Body)|Clauses], Goal, A_0, A, C) :-
    headunify(Head, Goal, Body, A_0, A_1),
    abs_prolog(Body, A_1, A_2, C),
    restrict(A_2, Head, Goal, A_Body),
    abs_prolog_clauses(Clauses, Goal, A_0, A_Clauses, C),
    join(A_Body, A_Clauses, A).
```

Figure 5.7: A collecting abstract Prolog interpreter. The additions are underlined.

5.4 Collecting Interpreter

Until now, we have ignored in the program the mapping from program points to abstract substitutions that is needed to make the abs_prolog/3 program correspond exactly to the *AbsProlog* specification given in Chapter 4. This is, however, simply an extra argument added to the entire interpreter; see Figure 5.7. We have added a call to a predicate update/4 which can update the mapping if necessary. This takes the goal Goal, two abstract substitutions A_{Goal} and A, and the extra mapping argument C which contains the data-structure (initially a free variable) that implements the mapping.

The correctness result given in the previous section allows us to insert calls to the update/4 predicate anywhere in the text of the abstract interpreter, but not within an implementation of a primitive.

5.5 Termination

Since the abstract interpreters we construct will be the basis of automatic program analysers it is important that they always terminate for all possible input programs.

We have seen in Chapter 4 that one approach to guaranteeing termination is to

incorporate some kind of bottom-up component in the abstract interpreter, assuming some finiteness property of the abstract domain. Here we incorporate an extension table mechanism into our interpreter shell. The resulting interpreter can be seen as an implementation of OLDT resolution. We choose this approach over the others discussed in Chapter 4 because we believe it is overall the simplest approach to Prolog abstract interpretation and a simple implementation is available in the work of Wærn [64].

5.5.1 Extension Tables

Here we describe the extension table mechanism we add to the abstract interpreter. It is based on the implementation of Wærn [64] who uses a suspension mechanism to express the bottom-up computation as a cyclic network of streams. A suspension mechanism in a Prolog system allows certain goals to be delayed until some instantiation state is reached by the variables in the goal. For example, the negation as failure predicate may be soundly implemented by delaying until the goal it is given is ground. Several current Prolog systems provide such a mechanism (such as NU-Prolog [73]), but it is also straightforward to construct an interpreter in Prolog incorporating suspension.

We introduced extension tables in Section 4.4.2. Dietrich's [24] implementation of extension tables for concrete Prolog globally stores a mapping from each call to the set of exits it has produced so far. The central idea behind extension tables is that if a call is subsumed by one already being executed then the exits of this previous call form a correct approximation of the exits of the subsumed call and so there is no need to recompute them. Dietrich implements a subsumed call by returning the current exit set stored in the extension table and then failing. All the exits, which will originate in other clauses, produced by the original (subsuming) call are added to its extension table entry. The entire interpreter is enclosed in an iterative loop so that with each iteration the exit set grows until a fixpoint is reached.

Because we have lifted our Prolog interpreter so that it operates from sets to sets, the extension table added to our concrete interpreter will consist of a mapping from sets of calls to sets of exits, and for the abstract interpreter it will be a mapping from an abstract call to a single abstract exit. That is, it mirrors the type of the interpreter as a whole.

In (our version of) a stream based implementation each entry in the extension table will be a mapping between an abstract call and a *stream* of improving approximations to the fixpoint representing the abstract exit. All calls are looked up in the extension table as they occur. If the call is not subsumed it is added to the table paired with its stream, initialized to the least element in the abstract domain; that is the stream $[\bot|\_]$. A subsumed call will return the (renamed) exit at the end of the stream—which represents the best approximation so far—and then, instead of failing, will suspend waiting on a new, larger exit to appear on the stream. The other clauses in the

predicate containing the suspended call can be executed, and the least upper bound of the results of these will eventually cause an exit to be computed for the original call which are added onto the stream.

Thus there is an iteration between a call producing a new, larger exit which wakes a subsumed call which in turn produces a new exit for the original call. Note that if this new exit is not larger than the previous one it is not added into the extension table. At this point the last exit in the stream is the fixpoint.

5.5.2 Example Extension Table Execution

As an example of the extension table mechanism described above, take the following program:

```
p(A) :- A=[].                    p([]).
p(A) :- A=[H|T], p(T).           p([H|T]) :- p(T).
?- p(X).                         ?- p(X).
```

Here we describe the abstract execution of the program assuming our groundness abstract domain (see Section 1.4) where g represents all ground Prolog terms and a represents all Prolog terms. These tokens are ordered $g \subseteq_A a$. An abstract substitution is a mapping from the local variables in a clause to either g or a. The ordering $\subseteq_A$ carries over to abstract substitutions. The initial call is:

$$\mathrm{p(X)}\ \langle \mathrm{X}/a \rangle$$

This call is not subsumed by any in the extension table (because it is empty) and so it is added to it paired to an empty exit stream (that is $[\bot|\_]$). The first clause of p/1 leads to the exit:

$$\mathrm{p(X)}\ \langle \mathrm{X}/g \rangle$$

The second clause of p/1 leads to a call:

$$\mathrm{p(T)}\ \langle \mathrm{T}/a \rangle$$

Looking this up in the extension table, it is subsumed by the first call and returns the last element on its exit stream (which is $\bot$) and then suspends on the remainder of the stream. Note that a suspended call will always immediately return some value; for recursive calls this will be $\bot$.

Now all the clauses of p/1 have been processed. The first clause has produced the exit $\langle \mathrm{X}/g \rangle$ on its output stream and the second clause has produced $\bot$. The least upper bound is therefore $\langle \mathrm{X}/g \rangle$ and this is inserted into the extension table as the next approximation to the fixpoint. At this point the top-down component has finished, and

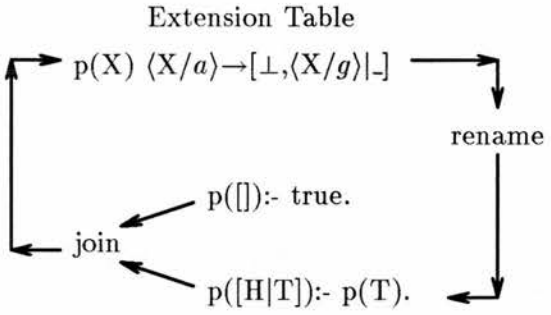


Figure 5.8: The state of the stream network after top-down execution has finished. The new substitution $\langle X/g \rangle$ has just been inserted into the extension table.

the bottom-up component is just about to start. The interpreter has set up a cyclic network of streams as shown in Figure 5.8.

The insertion of $\langle X/g \rangle$ into the extension table wakes the suspended call. The substitution is renamed to give:

p(T) ⟨T/g⟩

This is returned as the first exit of the suspended call. This propagates through the interpreter to give $\langle X/a \rangle$ as the next exit of the second clause; in a concrete interpreter X would be bound to [H]. The least upper bound is taken of the best exits of the two clauses—that is $\langle X/g \rangle$ from the first clause and $\langle X/a \rangle$ from the second—which is $\langle X/a \rangle$. This is inserted into the extension table as the second approximation to the fixpoint.

Again the suspended call wakes, but this time it leads to the same exit for the second clause and so the same substitution is produced as least upper bound. Since this is not larger then the last exit in the stream it is not itself inserted; rather the stream is closed by instantiating the end to [].

The final state of the extension table is:

p(X) ⟨X/a⟩→[⊥,⟨X/g⟩,⟨X/a⟩]

Therefore $\langle X/a \rangle$ is the fixpoint.

Now consider the slightly simpler program:

```
p(A) :- A=[].                    p([]).
p(A) :- A=f(T), p(T).            p(f(T)) :- p(T).
?- p(X).                         ?- p(X).
```

The final state of the extension table is:

p(X) $\langle X/a \rangle \rightarrow [\bot, \langle X/g \rangle]$

That is, the recursive clause added nothing to the result—p/1 always returns ground terms in its argument.

5.5.3 Implementation

Here we describe the modifications needed to the abstract Prolog interpreter given earlier (Section 5.2) to implement the extension table. The implementation is similar to that of Wærn [64] but improved in two ways:

- The streams contain elements which are monotonically increasing according to the $\subseteq_A$ ordering. This is more appropriate to our applications. This possible approach is described by Wærn only briefly in a concluding section and it does not seem to have been further investigated by her.
- Elements on streams are propagated in a different order. This improves the termination properties of the interpreter in some circumstances.

The abstract Prolog interpreter in Figure 5.4 is lifted to return a stream of abstract substitutions rather than a single abstract substitution (see Figure 5.9). Note that this interpreter does not include the program point mapping. There are a number of modifications:

- 'when' declarations (in the style of NU-Prolog [73]) are added to specify when calls should be suspended. Any predicate with a 'when' declaration is suspended until such a time as one of its 'when' declarations is satisfied. A 'when' declaration specifies some instantiation which a call must have and possibly some variables which must be instantiated. The declaration:

  ```
  :- abs_prolog(_, A, _) when A.
  ```

  states that the second argument must be instantiated. The 'ever' token can be used when there are no variables.

- The output of the interpreter (the third argument of abs_prolog/3) is raised from a single abstract substitution to a stream of abstract substitutions. This forces all predicates which process the output from abs_prolog/3 to accept a stream as input. Thus stream versions of the primitives compose/3, restrict/4 and join/3 are needed. For compose/3 and restrict/4 these are straightforward to implement, by wrapping the primitive with a predicate which waits on the input streams. Figure 5.10 gives an implementation of stream_compose/3, and stream_restrict/4.

```
:- abs_prolog(_, A, _) when A.
abs_prolog(true, A, [A]).
abs_prolog(L=R, A_0, [A]) :-
    bodyunify(L, R, A_0, A).
abs_prolog((Goal_L,Goal_R), A, S) :-
    abs_prolog(Goal_L, A, S_0),
    stream_abs_prolog(Goal_R, S_0, S).
abs_prolog(Goal, A, S) :-
    get_clauses(Goal, Clauses),
    trim(A, Goal, A_Goal),
    abs_prolog_clauses(Clauses, Goal, A_Goal, S_Clauses),
    stream_compose(A, S_Clauses, S).

abs_prolog_clauses([], _, _, [⊥]).
abs_prolog_clauses([(Head:-Body)|Clauses], Goal, A_0, S) :-
    headunify(Head, Goal, Body, A_0, A),
    abs_prolog(Body, A, S_Body),
    stream_restrict(S_Body, Head, Goal, S_Goal),
    abs_prolog_clauses(Clauses, Goal, A_0, S_Clauses),
    stream_join(S_Goal, S_Clauses, S).
```

Figure 5.9: Abstract Prolog interpreter over streams. Variables over streams begin with 'S'.

```
:- stream_compose(_, [], _) when ever.
:- stream_compose(_, [A|_], _) when A.
stream_compose(_, [], []).
stream_compose(A_Orig, [A_Goal|S_Goal], [A|S]) :-
    compose(A_Orig, A_Goal, A),
    stream_compose(A_Orig, S_Goal, S).

:- stream_restrict([], _, _, _) when ever.
:- stream_restrict([A|_], _, _, _) when A.
stream_restrict([], _, _, []).
stream_restrict([A_Body|S_Body], Head, Goal, [A|S]) :-
    restrict(A_Body, Head, Goal, A),
    stream_restrict(S_Body, Head, Goal, S).
```

Figure 5.10: Compose/3 and restrict/4 wrapped with stream handler.

```
stream_join([A_Goal|S_Goal], [A_Clauses|S_Clauses], [A|S]) :-
    join(A_Goal, A_Clauses, A),
    stream_join(S_Goal, A_Goal, S_Clauses, A_Clauses, S).

:- stream_join([], _, [], _, _) when ever.
:- stream_join([], _, [A|_], _, _) when A.
:- stream_join([A|_], _, [], _, _) when A.
:- stream_join([A|_], _, [B|_], _, _) when A and B.
stream_join([], _, [], _, []).
stream_join([], A_Goal, [A_Clauses|S_Clauses], _, [A|S]) :-
    join(A_Goal, A_Clauses, A),
    stream_join_one(S_Clauses, A_Goal, S).
stream_join([A_Goal|S_Goal], _, [], A_Clauses, [A|S]) :-
    join(A_Goal, A_Clauses, A),
    stream_join_one(S_Goal, A_Clauses, S).
stream_join([A_Goal|S_Goal], _, [A_Clauses|S_Clauses], _, [A|S]) :-
    join(A_Goal, A_Clauses, A),
    stream_join(S_Goal, A_Goal, S_Clauses, A_Clauses, S).

:- stream_join_one([], _, _) when ever.
:- stream_join_one([A|_], _, _) when A.
stream_join_one([], _, []).
stream_join_one([A_Goal|S_Goal], A_Clauses, [A|S]) :-
    join(A_Goal, A_Clauses, A),
    stream_join_one(S_Goal, A_Clauses, S).
```

Figure 5.11: Join/3 wrapped with stream handler.

The stream version of join/3, stream_join/3, waits on two streams. However, it waits for *both* of its input streams to produce a new element, unless one of them is finished. The code is given in Figure 5.11. The predicate stream_join/3 calls join/3 for the first elements on the streams (the streams are never empty) and then calls stream_join/5 which uses the extra arguments to store the last element on either stream, which is used when one of the streams in closed. If this happens the predicate stream_join_one processes the remainder of the open stream.

The effect of making stream_join/3 wait for both of its input streams to produce a new element or to be closed is to force an ordering on the way elements are propagated around the stream network. The ordering is the same as that described in the orginal formulation of OLDT resolution [71]. All elements are propagated through the extension table before any new elements may be produced by the suspended program.

This is illustrated by an example. Take the program used above to illustrate extension table execution modified by adding an extra clause:

```
p(A) :- A=[].
```

```
stream_abs_prolog(Goal_R, S_L, S) :-
    stream_abs_prolog(Goal_R, S_L, _, S).

:- stream_abs_prolog(_, [], _, _) when ever.
:- stream_abs_prolog(_, [A|_], _, _) when A.
stream_abs_prolog(_, [], S, S).
stream_abs_prolog(Goal_R, [A_L|S_L], _, [A_R|S]) :-
    abs_prolog(Goal_R, A_L, [A_R|S_R]),
    stream_abs_prolog(Goal_R, S_L, S_R, S).
```

Figure 5.12: The stream_abs_prolog/3 predicate

```
p(A) :- A=[H|T], p(T).
p(A) :- p(A).
?- p(X).
```

Both the second and third clauses suspend on the same extension table entry. There is a stream_join/3 predicate waiting on the output of these clauses. If it were to wait on either of these clauses then it is possible for just one clause to be woken and produce a new element which begins its propagation around the stream network. This will miss the new element produced by the other clause This means that the entire stream network may iterate more often than is necessary. Waiting on both inputs ensures that the elements from both clauses are joined together and that the fix point element is reached more quickly.

- abs_prolog/3 itself processes its own output (the third clause in Figure 5.9), so a stream version of this is given as stream_abs_prolog/3 (see Figure 5.12). This takes as input a stream of the exits from the left goal in a conjunction, executes the right goal with each of the elements on this stream and merges together each of the results. It does this by taking only the *first* element from the result stream of each call to abs_prolog/3. This is justified because if a new element appears on the result stream of the right goal, then one also appears on the result stream of the original left goal which makes this call to the right goal redundant.

The extension table itself can now be added to the interpreter. This is a single argument Et added to the program[1], together with calls to the predicates which maintain the extension table. The new version of abs_prolog is given in Figure 5.13. The two main predicates are:

[1]The extension table must be implemented as a single argument and not an accumulator pair. Accumulators depend on the standard Prolog ordering of the subgoals which may not occur with a delay based Prolog.

```
abs_prolog(Goal, A) :-
    abs_prolog(Goal, _, A, _).

:- abs_prolog(_, _, A, _) when A.
abs_prolog(true, _, A, [A]).
abs_prolog(L=R, _, A_0, [A]) :-
    bodyunify(L, R, A_0, A).
abs_prolog((Goal_L,Goal_R), Et, A, S) :-
    abs_prolog(Goal_L, Et, A, S_0),
    stream_abs_prolog(Goal_R, Et, S_0, S).
abs_prolog(Goal, Et, A, S) :-
    in_already(Goal, Et, A, S).
abs_prolog(Goal, Et, A, S) :-
    get_clauses(Goal, Clauses),
    trim(A, Goal, A_Goal),
    add_in(Goal, Et, A_Goal, S_Clauses),
    abs_prolog_clauses(Clauses, Goal, Et, A_Goal, S_Clauses),
    stream_compose(A, S_Clauses, S).

abs_prolog_clauses([], _, Et, _, [⊥]).
abs_prolog_clauses([(Head:-Body)|Clauses], Goal, Et, A_0, S) :-
    headunify(Head, Goal, Body, A_0, A),
    abs_prolog(Body, Et, A, S_Body),
    stream_restrict(S_Body, Head, Goal, S_Goal),
    abs_prolog_clauses(Clauses, Goal, Et, A_0, S_Clauses),
    stream_join(S_Goal, S_Clauses, S).

stream_abs_prolog(Goal_R, Et, S_L, S) :-
    stream_abs_prolog(Goal_R, Et, S_L, _, S).

:- stream_abs_prolog(_, _, [], _, _) when ever.
:- stream_abs_prolog(_, _, [A|_], _, _) when A.
stream_abs_prolog(_, _, [], S, S).
stream_abs_prolog(Goal_R, Et, [A_L|S_L], _, [A_R|S]) :-
    abs_prolog(Goal_R, Et, A_L, [A_R|S_R]),
    stream_abs_prolog(Goal_R, Et, S_L, S_R, S).
```

Figure 5.13: Abstract Prolog interpreter with extra extension table arguments Et. The additions are underlined.

```
in_already(Goal, Et, A, S) :- nonvar(Et),
    in_already_check(Goal, Et, A, S),

in_already_check(Goal, [Goal_ET-A_ET-S_ET|_], A, S) :-
    subsumes(Goal_ET, A_ET, Goal, A), !,
    stream_skip(S_ET, S_End),
    stream_renaming(S_End, Goal_ET, S, Goal).
in_already_check(Goal, [_|Et], A, S) :-
    in_already(Goal, Et, A, S).

stream_skip([A|S], [A|S]) :- var(S), !.
stream_skip([A], [A]) :- var(S), !.
stream_skip([_|S], S_End) :- stream_skip(S, S_End).

:- stream_renaming([], _, _, _) when ever.
:- stream_renaming([A|_], _, _, _) when A.
stream_renaming([], _, [], _).
stream_renaming([A_ET|S_ET], Goal_ET, [A|S], Goal) :-
    renaming(A_ET, Goal_ET, A, Goal),
    stream_renaming(S_ET, Goal_ET, S, Goal).
```

Figure 5.14: In_already/4 predicate implementing the extension table

```
add_in(Goal, Et, A, S) :- var(Et), !,
    Et=[Goal-A-S_ET|_],
    stream_filter(S, Goal, S_ET).
add_in(Goal, [_|Et], A, S) :-
    add_in(Goal, Et, A, S).

:- stream_filter([A|_], _, _) when A.
stream_filter([A|S], Goal, [A|S_ET]) :-
    stream_filter(S, Goal, A, S_ET).

:- stream_filter([], _, _, _) when ever.
:- stream_filter([A|_], _, _, _) when A.
stream_filter([], _, _, []).
stream_filter([A|_], Goal, A_0, []) :-
    subsumes(Goal, A_0, Goal, A), !.
stream_filter([A|S], Goal, A_0, [A|S_ET]) :-
    stream_filter(S, Goal, A_0, S_ET).
```

Figure 5.15: Add_in/4 predicate implementing the extension table

- in_already/4 (see Figure 5.14) which succeeds if some call is subsumed by one already in the extension table. One call a is subsumed by another b if $a \subseteq_A b$. The predicate subsumes/4 implements this ordering. If in_already/4 succeeds it first skips to the end of the stream, thereby getting the best approximation, and sets up a renaming filter so that any variables in subsuming calls are renamed to be consistent with the subsumed call. We assume the predicate renaming/4 implements this.

- add_in/4 (see Figure 5.15) which adds an entry into the extension table. A filter is placed in the input to ensure that the successive elements on the stream are increasing according to the $\subseteq_A$ ordering. When they do not increase the stream is closed.

The final collecting interpreter requires the collecting data structure to be added by adding an extra argument and calls to the update/4 predicate (see Section 5.4).

5.6 Summary

In this chapter, we have

- developed a framework for Prolog abstract interpretation which improves that of Wærn [64] by reaching the fixpoint after fewer iterations in general.

- developed some convenient conditions which an abstract domain must satisfy in order to result in correct and terminating abstract interpreters. The conditions are comparable to those used by Bruynooghe [7].

- presented an implementation of the framework by using Prolog as the description language.

Chapter 6

An Abstract Domain to Infer Inclusion and Sharing

6.1 Introduction

In Chapter 3 a number of programming techniques were defined in terms of inclusion and sharing relationships between the local variables in clauses. Here an abstract domain for inferring these relationships is developed. As we described in Chapter 3, this enables us to detect instances of techniques in Prolog code.

Inclusion and sharing relationships reflect the effects of the unifications which occur during the execution of a program. As we have seen in Chapter 4, to construct an abstract interpreter to infer certain information a concrete interpreter is needed which makes this information explicit. We therefore need an instrumented concrete Prolog interpreter which makes the effects of unifications explicit. We will describe this as a concrete domain which instantiates the framework given in Chapter 5. We then develop the abstract domain.

6.2 Concrete Domain

In Chapter 3 we defined the inclusion and sharing relations using a notion of equality over a special form of terms called *indexed* terms. Here we develop an instrumented description of concrete Prolog which allows the equal terms to be detected at any point in the computation. This forms the basis for an abstract domain to detect inclusion and sharing. We first describe the representation for substitutions in this concrete Prolog, and then the primitives over the substitutions. This concrete domain instantiates the interpreter framework developed in Chapter 5.

The concrete Prolog is based on *indexed* terms as introduced in Section 3.10. There an index is some arbitrary number. Here an index consists of a pair (constructed with

':') of:

1. A token which is assigned statically to every occurrence of each function symbol in the program so that it uniquely identifies this occurrence.

2. A token which is assigned dynamically so that it uniquely identifies each predicate call during the execution.

An *indexed substitution* is a mapping from the local (textual) variables in a clause to an indexed term. Free variables are represented by a special token *Var* which also have an index.

To illustrate, consider the program:

```
?- p(X).
p(A) :- A=f(B), q(B).
q(C) :- C=f(D), D=a.
```

we first assign some indices to the function symbols:

```
?- p(X).
p(A) :- A=f1:_(B), q(B).
q(C) :- C=f2:_(D), D=a3:_.
```

For the moment the right hand side of the index pair, which is assigned dynamically, is left as an underscore '_'. Note that the two occurrences of f/1 get different indices. At the beginning of the execution of p/1's body the substitution is:

$$\langle\ \mathrm{A}/Var_{A:1}\ \mathrm{B}/Var_{B:1}\ \rangle$$

and at the end of the body:

$$\langle\ \mathrm{A}/\mathrm{f}_{1:1}(\mathrm{f}_{2:2}(\mathrm{a}_{3:2}))\ \mathrm{B}/\mathrm{f}_{2:2}(\mathrm{a}_{3:2})\ \rangle$$

The first half of the index is the static token. The variables use the name of their textual variable as a token. This assumes that variable names uniquely identify variables in predicates rather than clauses. The function symbols use a number as a token. The second half of the index is the call count[1].

Using this representation for substitutions, aliased variables are bound to the same *Var* term. For example, after the unification X=Y the substitution will be:

$$\langle\ \mathrm{X}/Var_{X:1}\ \mathrm{Y}/Var_{X:1}\ \rangle$$

We will often not give indices explicitly but denote them using the letters $i, j, k \ldots$.

We can extract the inclusion and sharing relations from indexed substitutions as follows:

[1]Indices contain a dynamic component so they need to be mapped in the renaming operation assumed by our framework. Note that this applies to both variables and function symbols.

- A variable X includes another Y if the set of indices in Y's term is a subset of the set of indices in X's term.
- A variable X shares with another Y if the set of indices in Y's term is not disjoint with the set of indices in X's term.

The concrete interpreter framework (see Chapter 5) needs to operate over *sets* of indexed substitutions where each substitution is over the same set of variables. We can extend the above definitions to extract the inclusion and sharing relations from sets of indexed substitutions as follows:

- A variable X includes another Y in a set of indexed substitutions if X includes Y in any member of the set.
- A variable X shares with another Y in a set of indexed substitutions if X shares with Y in any member of the set.

We now define the primitives for the concrete domain.

6.2.1 Restriction

Restriction tidies up the set of indexed substitutions C at the end of a clause body execution and renames the variables in terms of the original goal G. For example, if C is:

$$\{\langle \mathrm{A}/\mathrm{f}_i(\mathit{Var}_k)\ \mathrm{B}/\mathit{Var}_k\ \mathrm{C}/\mathrm{f}_j(\mathit{Var}_k)\rangle\},$$

the head of the clause H is p(A,B) and G is p(X,Y), then the result of restriction is:

$$\{\langle \mathrm{X}/\mathrm{f}_i(\mathit{Var}_k)\ \mathrm{Y}/\mathit{Var}_k\rangle\}.$$

The procedure is defined:

> For each indexed substitution I in C,
> remove all variable mappings from variables not in H,
> rename any remaining variables with variable in corresponding
> argument of G.
> Gather resulting indexed substitutions into the output set.

6.2.2 Join

Join takes the results of two clause executions (that is, two sets of indexed substitutions) over the same set of variables and merges them by finding their least upper bound in the ordering over the concrete domain; this is, the union of the two sets.

6.2.3 Trim

Trim removes from a set of substitutions all parts not relevant to a goal; that is, all mappings not involving a variable in that goal.

6.2.4 Head Unification

Head unification takes a set of indexed substitutions C just before a goal G, and constructs a set of indexed substitutions suitable for executing a clause with head H and body B. For example, if C is:

$\{\langle \mathrm{X}/\mathrm{f}_i(\mathit{Var}_j)\ \mathrm{Y}/\mathit{Var}_j\rangle\}$,

with G as p(X,Y) and the clause is:

p(A,B) :- q(A,C), r(C,B).

then the output of head unification, which is the initial substitution for executing the body, is:

$\{\langle \mathrm{A}/\mathrm{f}_i(\mathit{Var}_j)\ \mathrm{B}/\mathit{Var}_j\ \mathrm{C}/\mathit{Var}_k\rangle\}$

The procedure is defined:

For each indexed substitution I in C,
  rename variables with the variables in corresponding arguments of H,
  for each new variable in B,
    add a mapping to a *Var* term with a new index.
Gather resulting indexed substitutions into the output set.

6.2.5 Body Unification

Body unification unifies two terms L and R in the context of a set of indexed substitutions C. To ensure that equality constraints are propagated correctly body unification is implemented in a slightly nonstandard way. To explain this we first give a description of standard unification (but excluding the occur check) and then add to it our modifications. The standard description of unification of two terms is:

For each indexed substitution I in C,
  Replace all the variables in L and R with their values in I.
  Check corresponding subterms of resulting index terms
    If they have the same function symbol (ignoring indices)
      go to next subterm
    If one is *Var*
      replace all occurences of it with the other side
      and go to next subterm
    Otherwise fail.
Gather resulting indexed substitutions into the output set.

To implement our notion of indexed terms equality, when two function symbols are successfully unified they must not only be equal but they should have the same index. To ensure this, it is simply necessary to reindex one of the two sides, together with all other occurrences of it in the substitution. Our concrete unification is a simple modification to the above procedure (the additions are underlined):

For each indexed substitution I in C,
  Replace all the variables in L and R with their values in I.
  Check corresponding subterms of resulting index terms
    If they have the same function symbol (ignoring indices)
      reindex all occurrences of the symbol on the left with the index on the right.
      go to next subterm
    If one is *Var*
      replace all occurrences of it with the other side
      and go to next subterm
    Otherwise fail.
  Gather resulting indexed substitutions into the output set.

The choice of reindexing the left with the right is arbitrary.

For example, the result of the query:

?- X=f_i(Y), Z=f_j(Y).

is:

{⟨ X/f_i(Var$_k$) Y/Var$_k$ Z/f_j(Var$_k$) ⟩},

and the result of:

?- X=f_i(Y), Z=f_j(Y), X=Z.

is:

{⟨ X/f_j(Var$_k$) Y/Var$_k$ Z/f_j(Var$_k$) ⟩}.

6.2.6 Composition

The composition primitive implements (unifying) composition by merging the results from a goal C with the substitutions before the goal C_B. For example, if before the goal the set of substitutions is:

{⟨X/f_i(Var$_j$) Y/f_k(Var$_l$,Var$_m$) Z/Var$_m$⟩}

and the result from the goal is:

{⟨X/f_i(Var$_j$) Y/f_k(f_i(Var$_j$),a_n)⟩},

then the result of compose is:

$\{\langle X/f_i(Var_j)\ Y/f_k(f_i(Var_j),a_n)\ Z/a_n\rangle\}$

The procedure is defined as:

For each indexed substitution I in C,
for each indexed substitution I_B in C_B,
set I_O to I_B
for each local variable V in I
extract all the changed terms by comparing V's values
(using indexed term equality) in I_O and in I
replace all occurrences of these changed terms in I_O with corresponding I.
return final I_O.
Gather resulting indexed substitutions into the output set.

In the above example, the changed terms in Y's value before and after the goal are Var_l which is changed to $f_i(Var_j)$ and Var_m which is changed to a_n. Note that indexed terms can change as well as variables.

6.3 The Abstract Domain

Inclusion and sharing relationships arise when some indexed term appears in the binding of more than one local variable. This forms the basis of our abstract domain. Take a single indexed substitution. For each indexed (sub)term it contains, form the set of the local variables of which it is a subterm in the substitution. Such a set is called the *term descriptor* for the term. For short we write this as *Td*. So for example, the *Td* for the term a_j in the substitution:

$\langle X/f_i(a_j)\ Y/a_j\ Z/f_k(a_j)\rangle$

is the set {X,Y,Z} and for the term $f_i(a_j)$ it is {X}. We can then form the set of term descriptors in an entire indexed substitution. In the above example, this is {{X,Y,Z},{X},{Z}}. We call this set of *Tds* a *substitution descriptor* for the substitution. For short we write this as *Sd*. Given an *Sd* it is possible to infer inclusion and sharing relationships between variables:

- A includes B if all term descriptors containing B also contain A. In the above *Sd*, the only term descriptor containing Y is {X,Y,Z} and so X (and Z) must include Y.
- A shares with B if A and B appear together in at least one term descriptor. In the above, the node descriptor {X,Y,Z} means that X and Z share, but neither includes the other because they both appear in other *Tds*.

An abstract substitution may contain *Tds* which do not correspond to any term created in any possible concrete substitution. Thus the inclusion and sharing relationships cannot be detected with certainty; that is, we can only say that two variables *possibly* share or include. This is, however, all we need for our application.

An abstract domain based on *Sds* apparently provides a simple and elegant way of capturing inclusion and sharing between variables. However, it is necessary to provide more information in an abstract substitution to allow sharing relationships to be propagated precisely during an abstract execution.

To illustrate, consider unifying two terms L and [H|T] given an *Sd* {{L},{H},{T}}. Each of L, H and T are bound to arbitrary terms about which we know nothing apart from that they are completely separate (because L, H and T appear in separate *Tds*). What can be said about the *Sd* after a successful unification?

- The *Td* {L} should still be present because it would result from L being unified with the ./2 part of [H|T].
- The *Td* {H,L} should be present because it would result from terms in the result of unifying the left subterm of L with H.
- Similarly for {T,L}.
- The *Td* {H,T,L} should be present because it can occur if the left and right subterms of L share. For example, this would occur if L was bound to the term $[a_j|a_j]$.

Thus the new *Sd* is {{L},{H,L},{T,L},{H,T,L}}. Note that the terms bound to H and T may share. If, however, it were known that the left and right subterms of L did not share then the new *Sd* would be {{L},{H,L},{T,L}} and it would be possible to state that H and T were independent. This occurs if we know that the L term is a tree; that is, no sibling subterms contain overlapping sets of indices.

This motivates the need for a separate component in an abstract substitution which tells which local variables may not be a tree. This can simply be the subset of the local variables for which 'treeness' is not known. We call this set the *non-tree descriptor* of the substitution. For short we write this as *Nd*. Note that a new free variable is certainly a tree and need not be added to this set.

Note that our framework for abstract interpretation described in Chapter 5 requires that an abstract substitution represents a set of indexed substitutions, and that the abstract domain contains a unique bottom element $\perp$. To deal with this, we extend the informal description above in the following section where we formally describe the relationship between the concrete and the abstract domains.

We represent an abstract substitution by a pair formed from the token *subst*. The first half of this pair is an *Sd* and the second half is an *Nd*. We give some example abstract substitutions with an informal description of what they represent:

- *subst*({{X},{X,Y}}, {}). Here Y is included in X, that is Y is bound to some part of X. Such a substitution arises after a call to the usual member/2 predicate where X is the list and Y is the element.
- *subst*({{X},{X,Y}}, {X}). This is similar except X is not known to be a tree, but Y is. This can occur if the list contains repeated elements.
- *subst*({{X},{X,Y},{Y}}, {}). Here X and Y share. But there are parts of both which are independent. Such a substitution arises after a call to a maplist/2 predicate.

6.4 Abstraction and Concretization

We define precisely the relationship between the concrete and the abstract domains by providing the abstraction *abs* and concretization *conc* functions introduced in Chapter 4.

6.4.1 Abstraction

The input to *abs* is a set of indexed substitutions C over the same set of local variables. The procedure is:

If C is empty return $\perp$
else
  set both s and n to the empty set.
  For each I in C,
    for each separate indexed function symbol
      construct the bag of the local variables it appears in.
      Add variables which appear more than once in the bag to n.
      Make a set out of the bag and add it to s.
Return *subst*(s, n).

Note that in the above 'add' means addition using set union.

6.4.2 Concretization

The input to *conc* is an abstract substitution A and it returns a set of indexed substitutions. It can be defined in terms of *abs*:

If A is $\perp$ return the empty set
else
$A = subst(s, n)$
return $\{x \mid abs(x) = subst(s', n') \wedge s' \subseteq s \wedge n' \subseteq n\}$

6.5 Termination

The above abstract domain guarantees termination of the framework interpreter because it is finite. This is because each possible abstract substitution is constructed only out of a finite set of local variables.

To place it within our framework for abstract interpretation (see Chapter 5) we need to define a subsumption ordering on the abstract domain and provide a predicate subsumes/4 which can compare two abstract substitutions within this ordering.

Because of the finiteness of the domain, one possible definition of subsumption is substitution equality. The following definition, however, can make the abstract interpreter terminate quicker. An abstract substitution A_g subsumes another A_l over the same set of variables if A_g's concretization is a superset of A_l's concretization. This holds if:

$$A_{lSd} \subseteq A_{gSd}$$

and

$$A_{lNd} \subseteq A_{gNd}$$

where

$$A_g = subst(A_{gSd},\ A_{gNd})$$

and

$$A_l = subst(A_{lSd},\ A_{lNd}).$$

In addition, the abstract substitution $\perp$ is subsumed by everything and subsumes nothing other than itself.

It is also possible to define a least upper bound function in this ordering. The least upper bound of two abstract substitutions A_1 and A_2 is:

$$subst(A_{1Sd} \cup A_{2Sd},\ A_{1Nd} \cup A_{2Nd}).$$

If either abstract substitution is $\perp$ then the other is returned.

This subsumption check assumes that the abstract substitutions are over the same set of variables. If not, it is necessary to canonicalize each one so that they are comparable.

6.6 Primitive Operations

We now give a description of the abstract versions of the primitives needed by the framework.

6.6.1 Restriction

Restriction tidies up the substitutions at the end of a clause body execution and renames the variables in terms of the original goal. For example, if the substitution A is:

subst({{A},{A,B},{A,C},{C}}, {}),

the head of the clause H is p(A,B) and the goal G is p(X,Y) the resulting substitution is:

subst({{X},{X,Y}}, {}).

The procedure is defined:

If A is $\perp$ then return $\perp$
else
  return a new substitution formed by
  intersecting each *Td* and the *Nd* in A with the set of variables in H,
  then renaming remaining variables with the variables in
    corresponding arguments of G.

This procedure is justified because completely removing a variable from a substitution does not affect the inclusions and sharings involving other variables.

6.6.2 Join

Join takes the results of two clause executions (that is, two abstract substitutions) and merges them. This merge is the least upper bound in the subsumption ordering over the abstract domain. For example, the join of the two substitutions

subst({{X},{Y}}, {X})

and

subst({{X,Y}}, {Y})

is

subst({{X,Y},{X},{Y}}, {X,Y}).

The procedure is defined:

If either substitution is $\perp$ then return the other
else
  return a substitution whose *Sd* is the union of the *Sds* of
    the input substitutions,
  and whose *Nd* is the union of the *Nds* of
    the input substitutions.

6.6.3 Trim

Trim removes from an abstract substitution A all parts not relevant to a goal G. For example, if the substitution is:

subst({{X},{X,Y},{X,Z},{Z}}, {}),

and the goal is p(X,Y) the resulting substitution is:

subst({{X},{X,Y}}, {}).

The procedure is defined:

If A is $\perp$ then return $\perp$
else
  return a substitution formed by
  intersecting each *Td* and the *Nd* in A
  with the set of variables in G.

6.6.4 Head Unification

Head unification takes an abstract substitution A just before a goal G, and constructs an abstract substitution suitable for executing a clause with head H and body B. For example, if A is:

subst({{X},{X,Y},{Y}}, {X}),

with G as p(X,Y) and the clause is p(A,B) :- q(A,C), r(C,B). then the initial substitution for executing the body is:

subst({{A},{A,B},{B},{C}}, {A}).

The procedure is defined:

If A is $\perp$ return $\perp$
else
  return a substitution formed by
  renaming all the variables in A by pairing the variables
  in H with G, and then
  for all variables V in B but not H add also
    the Td $\{V\}$.

6.6.5 Body Unification

Body unification returns the substitution after a successful unification of the two sides L and R, given some initial substitution A. We motivate the procedure below by considering concrete body unification. Concrete body unification does two things: first, it traverses each side of the unification checking terms, and second, it replaces terms throughout the substitution so that these sides are made the same. Consider how replacing a term throughout a concrete substitution would affect an abstract substitution:

1. The Td representing the term gets replaced by the union of it and the Td representing the replacing term.

2. If either side of the unification is not a tree then a replacement term may itself get replaced. This means that several Tds may be unioned together, even Tds from the same side.

3. If the term itself is not a tree then the variable, part of whose value is being replaced, becomes nontree.

4. If a variable's value already contains the replacing term then that value may cease to be a tree, and so the variable may become nontree.

The above observations motivate the following procedure for abstract body unification.

Since there is no information about the structure of the two sides it is necessary to consider replacing each Td from one side with each Td from the other side. This is not quite correct because of the second point above. If one side is possibly a nontree (which means that the term under consideration may occur several times), then it is necessary to consider replacing each Td with every *subset* of the Tds on the other side.

The top level of the procedure is as follows:

If A is $\perp$ return $\perp$
else
  Set S to the set of *Tds* of A which are disjoint
    from the variables in L and R.
  Set N to the *Nd* of A.
  Set P_L to the *Tds* on the left.
  Set P_R to the *Tds* on the right.
  For each pair (l,r) in $P_L \times P_R$
    update S using (l,r)
    update N using (l,r)
  Return *subst*(S, N).

In the above, after handling the $\perp$ case, an initial *Sd* S is formed by extracting out all *Tds* in A's *Sd* which do not contain any variable in either L or R, and an initial *Nd* N is set to A's *Nd*. Then two sets P_L and P_R are constructed each representing all the possible replacements from each side. The cartesian product of these sets is iterated through, evaluating the effects of replacing the corresponding terms updating S and N. The procedures to do this updating are described below. Finally, a substitution formed out of the final values of S and N is returned.

The sets P_L and P_R are constructed using the following procedure which is called once for each side. We describe here the procedure assuming it is constructing P_L; it is similar for P_R, with L and R reversed. The inputs are the term on the left L, the term on the right R and Sd_L the *Sd* of A with all *Tds* involving only variables not in L removed:

if L is a textual variable then
  if R contains any variable which is nontree then
    return the powerset of Sd_L with the empty set removed
  else return the powerset of Sd_L with all nonsingleton members removed
else if R contains any variable which is nontree then
    return the powerset of $Sd_L \cup \{\}$ with the empty set removed
  else return the powerset of $Sd_L \cup \{\}$ with all nonsingleton members
  removed

The above first adds the empty *Td*—that is, the empty set {}—if L is not a textual variable. This represents the parts of the term in the program. Then the procedure returns the powerset of the resulting set of *Tds*, with certain members removed. If R is a tree then all the nonsingleton members may be removed because the situation dealt with in the second point above does not arise. If R is not known to be a tree multiple replacements may occur. In either case the empty set is removed because each replacement involves at least one term.

We now give procedures for updating the sets S and N given members of P_L and P_R named l and r respectively. Note that both l and r are sets of *Tds*. We denote the union of all the members of l and r as $l_\cup$ and $r_\cup$. The following procedure updates S:

S_i is S_{i-1} with a new *Td* formed by unioning $l_\cup$ and $r_\cup$

This follows from the first point about concrete unification; the result of replacing terms is the union of the *Tds* which represent them.

The procedure for updating N is more complex. It follows from the third and fourth points about concrete unification:

N_i is N_{i-1} unioned with
the intersection of $l_\cup$ and $r_\cup$ and
$l_\cup$ if $r_\cup$ is a subset of N_{i-1} and
$r_\cup$ if $l_\cup$ is a subset of N_{i-1} and

The first addition is motivated by the fourth point about concrete unification. A variable's value possibly contains the replacing term (or some subterm) if it contains a *Td* including the same variables. The last two additions are motivated by the third point, since a *Td* is nontree if it is a subset of the current *Nd* component.

To illustrate the above procedure we now give some examples. The first is the unification X=Y with A as:

subst({{X,Z},{Y},{Y,W}}, {}).

All the *Tds* in the substitution are involved in the unification so S_0 is assigned to {}. The sets P_L and P_R are {{{X,Z}}} and {{{Y}},{{Y,W}}} respectively, which means there are two (l,r) pairs, ({X,Z},{Y}) and ({X,Z},{Y,W}). The result of the unification is:

subst({{X,Z,Y},{X,Z,Y,W}}, {}).

The *Td* {X,Z,Y} arises from unioning {X,Z} and {Y}, and {X,Z,Y,W} from unioning {X,Z} and {Y,W}.

The above example involved unifying tree values. Now consider the unification X=[H|T] where A is:

subst({{X},{H},{T}}, {X})

Here the value of X is nontree. This means that the set P_L is {{{X}}} and the set P_R is:

{{{}},{{H}},{{T}},
{{},{H}},{{},{T}},{{H},{T}},
{{},{H},{T}}}.

This is the powerset of {{},{H},{T}} with the empty subset removed. The result of the unification is:

subst({{X},{X,H},{X,T},{X,H,T}}, {X,H,T}).

The *Nd* set is built when the {X} is replaced by any *Td* on the other side, since there may be multiple occurrences of the corresponding term.

Note that the procedure given above is not optimal; that is, it is not as precise as possible. For example, the procedure only sees if the whole of one side is possibly a nontree, whereas abstract substitutions contain sufficient information to decide whether a particular *Td* is nontree. However, the extra complexity to the procedure is not worth the gain in precision.

6.6.6 Composition

Composition takes the abstract exit substitution E just after a subgoal G and the original abstract substitution O just before that subgoal and propagates the effects of the subgoal into an abstract substitution for executing the rest of the clause. Note that the exit substitution involves only the variables in the subgoal.

We motivate the procedure below by considering concrete (unifying) composition. Concrete composition does two things: first, it finds terms in the exit substitution which are different from corresponding terms in the original substitution, and second, it replaces terms throughout the substitution.

In terms of its effects on an abstract substitution, the replacement of terms is the same as that in body unification above. What is different is the way in which possible replacements are constructed. A *Td* x in E_{Sd} could have been formed from a subset s of the *Tds* in O_{Sd} each of which are not disjoint with x and which collectively contain all the variables in x. The *Td* x was caused by a unification involving any pair of subsets of s. The toplevel of the compose procedure is:

```
If E is ⊥ return ⊥
else
   Set S0 to the subset of O_Sd which are not involved in G.
   Set N0 to O_Nd.
   For each Td x in E_Sd
      Form the subset s of all Tds in O_Sd which could have caused x
      For each pair (l,r) of subsets of s
         update S_i as in body unification
         update N_i as in body unification
   Return subst(S_n, N_n).
```

The following example occurs during the execution of the member/2 predicate with the recursive call member(X,T):

```
member(X, L) :- L = [X|T].
member(X, L) :- L = [H|T], member(X, T).
```

The original substitution is:

subst({{X},{L},{L,H},{L,T}}, {})

and the exit substitution is:

subst({{X,T},{T}}, {}).

The correct result is:

subst({{X,L,T},{L},{L,H},{L,T}}, {})

6.7 Example Results

We have implemented the abstract domain developed above. Here we give the results of an analysis of some sample predicates.

The first example is the standard quick reverse predicate. The code, which is in our normal form (see Section 1.6), is annotated with the abstract substitution at the neck of the clause and after every subgoal:

```
?- Y=[], qr(X,Y,Z).
qr(A,B,C):-  : subst({{A},{B},{C}}, {})
   A=[], : subst({{A},{B},{C}}, {})
   B=C. : subst({{A},{B,C}}, {})
qr(A,B,C):-  : subst({{A},{B},{C},{D},{E},{F}}, {})
   A=[D|E], : subst({{A},{A,D},{A,E},{B},{C},{F}}, {})
   F=[D|B], : subst({{A},{A,D,F},{A,E},{B,F},{C},{F}}, {})
   qr(E,F,C). : subst({{A},{A,C,D,F},{A,C,E},
         {A,E},{B,C,F},{C},{C,F}}, {})
```

These annotations were obtained on the abstract interpreter by collecting the least upper bound of the current abstract substitutions when execution passed through the various program points.

We can process this into a form suitable for detecting techniques by extracting the sharing ($\sim$) and inclusion ($\gg$) relations. We require only the relations caused by each subgoal, though all the previously inferred relations still hold. Note that if two variables are in the inclusion relation then they are also in the sharing relation. Similarly, every variable both includes and shares with itself. These extra elements in the relations are not made explicit in the annotations.

```
qr(A,B,C) :-
    A=[],
    B=C. : B≫C, C≫B
qr(A,B,C) :-
    A=[D|E], : A≫D, A≫E
    F=[D|B], : A∼F, F≫B, F≫D
    qr(E,F,C). : A∼C, C∼E, C≫B, C≫D, C≫F
```

The technique detection procedure described in Chapter 3 can infer (at least) the following labels from these annotations:

```
qr(A:Up, B:Acc, C:AccResult) :-
    A=[],
    B=C. : B≫C, C≫B
qr(A:Up, B:Acc, C:AccResult) :-
    A=[D|E], : A≫D, A≫E
    F=[D|B], : A∼F, F≫B, F≫D
    qr(E,F,C). : A∼C, C∼E, C≫B, C≫D, C≫F
```

A more complex example is the flatten list predicate, for which we give only the techniques detecting form:

```
?- Z=[], flatten(X,Y,Z)
flatten_dl(A,B,C) :-
    A=[],
    B=C. : B≫C, C≫B
flatten_dl(A,B,C) :-
    B=[A|C], : B≫A, B≫C
    A\=[],
    A\=[_|_].
flatten_dl(A,B,C) :-
    A=[D|E], : A≫D, A≫E
    flatten_dl(D,B,F), : A∼B, B∼D, B≫F
    flatten_dl(E,F,C). : A∼F, B∼E, E∼F, B≫C, F≫C
```

In the last clause, the second and third arguments can be labelled with the difference list technique:

```
flatten_dl(A,B:TypicalDLUp,C:TypicalDLDown) :-
    A=[D|E], : A≫D, A≫E
    flatten_dl(D,B,F), : A∼B, B∼D, B≫F
    flatten_dl(E,F,C). : A∼F, B∼E, E∼F, B≫C, F≫C
```

6.8 Comparison

The abstract domain we have described in this chapter can be compared with Bruynooghe's domain for the compile time garbage collection application (see Sec-

tion 4.3.4). Although it assumes a different concrete description of Prolog—namely, a real Prolog implementation in his case and a novel equality constraint description in ours—they both infer similar kinds of information. The differences are as follows:

- The complexity of our domain is potentially higher because whereas Bruynooghe's domain is based on sets of pairs of variables, ours is based on sets of sets of variables. Thus an abstract interpreter based on our domain may take more iterations to reach a fixpoint. It is difficult to compare the complexity of the implementation of the primitives because Bruynooghe does not give a complete description.

- The information inferred by our domain is more precise. This is because of the set of sets representation of the inclusion and sharing, and because of the addition of the *Nd* set.

We have seen above (see Section 6.3) how the *Nd* set improves the quality of the information. The following example shows how our domain is more precise because the set of sets representation individually describes the subterms in the value of a variable, rather than the value being described as a whole. Consider the execution of the following clause, assuming that the two variables A and C are initially independent of each other:

p(A,C) :- A=f(X), B=f(X,Y), C=f(Y).

Assuming Bruynooghe's abstract domain (but ignoring the type information it includes) the substitution after each unification will be as follows:

{ X PART A }

{ X PART A
  X PART B, Y PART B, A AL B }

{ X PART A,
  X PART B, Y PART B, A AL B,
  Y PART C, C AL B, C AL A }

This means that the substitution after the execution of the body is

{ C AL A }

that is the variables A and C possibly share.

With our abstract domain the corresponding substitutions are as follows:

subst({{A},{A,X}}, {})

subst({{A},{A,B,X},{B},{B,Y}}, {})

subst({{A},{A,B,X},{B},{B,C,Y}, {C}}, {})

and the final substitution is:

subst({{A}, {C}}, {})

so the variables certainly do not share.

On the other hand, Bruynooghe qualifies his sharing and inclusion information with some typing annotations. This allows better quality information in some circumstances (though this does not affect the above example).

The precision problems with Bruynooghe's domain have been overcome in more recent work [55]. This however incorporates a full type inferencing abstract domain and is therefore considerably more complex.

Note that it is possible to adapt our domain for the compile time garbage collection application of Bruynooghe, though this would involve reimplementing the primitives we have given since Bruynooghe's domain is based on an instrumented concrete description of Prolog which corresponds to real Prolog implementations.

The idea behind the 'set of sets of variables' representation we have used for the graph descriptor is taken directly from the work of Jacobs and Langen [34] on inferring variable aliasing (see Section 4.3.3). The semantics of the domain, and so the implementation of the primitives, is of course quite different.

A prototype implementation of the abstract domain has been implemented based on the code for the framework given in Chapter 5 and the procedures described here. It is capable of analysing predicates such as the examples given. The results of the analysis are precise enough to allow the techniques contained in the code to be detected. Note that because the abstract domain infers only uncertain information—for example, X and Y possibly share—the analysis can only suggest the occurrence of techniques.

Inclusion and sharing relations are sufficient for describing a number of useful programming techniques. However, as we discussed in Section 3.11, we require extensive type information to be able to describe a larger range of programming techniques. In the following chapter, we present an abstract domain capable of such type inference.

6.9 Summary

In this chapter, we have

- developed an instrumented concrete domain for Prolog making the effects of unifications explicit.
- developed an abstract domain for inferring inclusion and sharing relationships and described its implementation.

- given some examples of the information the abstract domain infers and the techniques that can then be detected.

Chapter 7

An Abstract Domain to Infer Type Information

7.1 Introduction

In Chapter 6 we developed a simple domain which inferred inclusion and sharing relationships between variables. This enabled a range of Prolog programming techniques to be detected. In Chapter 3 we showed the limitations of a technique classification based only on inclusion and sharing, and suggested that any improved classification would involve making use of certain type information.

Here we outline an abstract domain which infers type information which we believe is suitable for defining an extended range of techniques. Its main feature is its ability to represent *relationships* between the data-structures bound to different variables. However, the abstract domain is quite complex and we only sketch the central ideas rather than specify it fully. It can therefore only be regarded as tentative.

7.2 Motivation

Inclusion and sharing only capture simple structural relationships between variables. Here we consider the kind of information needed for more complex structural relationships between data-structures. Consider again the quick sort program:

```
quicksort(A, B) :- C=[], qsort(A, B, C).

qsort(A, B, C) :- A=[D|E],
    partition(D, E, F, G),
    H=[D|I], qsort(F, B, H),
    qsort(G, I, C).
qsort(A, B, C) :- A=[], B=C.
```

```
partition(A, B, C, D) :-
   B=[], C=[], D=[].
partition(A, B, C, D) :-
   B=[E|F], C=[E|G], A @≥ E, partition(A, F, G, D).
partition(A, B, C, D) :-
   B=[E|F], D=[E|G], A @< E, partition(A, F, C, G).
```

In Section 3.11 we suggested that the second clause of qsort/3 was an instance of some *Split* list technique which had the following schema:

```
head(L:Split) :-
   subgoal:L ▷S, L ▷G &
   subgoal(S) &
   subgoal(G)
```

Where ▷ was a new relation named 'partof' which was similar to inclusion. This is a crude approximation to what is really occurring in qsort/3. The following are other, successively stronger, ways of describing the relationship between the variables L, S and G:

- G, S and L are lists.
- G and S are both sublists of L.
- G and S do not contain common elements.
- Each element in L is used exactly once, appearing in either G or S.

Ideally, therefore, we not only need to be able to make statements about the form of the data-structures a variable may be bound to, but also statements about relationships between the data-structures bound to different variables.

The type language developed below allows data-structures to be described in such a way that these relationships may be inferred. The language attempts to capture the effects of each clause execution and combine these effects together into a representation of the entire program. The type language is based on two kinds of such combining constructs:

Choices: Because clauses act as alternative ways of executing a call it must be possible to represent choices for the value of each variable in the call.

Recursions: A large data-structure is always constructed incrementally by the combined effects of a number of clause executions. With a recursive data-structure clauses may be executed more then once. In an abstract substitution we need to represent recursive data-structures finitely by forming notional 'loops' in terms.

7.3 Concrete Domain

An abstract domain needs to be based on a concrete domain; that is, a concrete description of Prolog. The concrete domain assumed here is based on the instrumented concrete domain described in Section 6.2 with two modifications:

- the form of the indices of variables and function symbols;
- the way unification reindexes function symbols.

In Section 6.2 we took an index to be a pair of tokens, one of which uniquely identifies each occurrence of each function symbol, and the other which uniquely identifies each predicate call during the execution. In the abstract domain developed below we need to be able to construct indices so that certain relationships between terms are maintained. In particular, we need to be able to ensure that we can identify terms constructed in earlier predicate calls. To do this we need a more complex data structure to represent the predicate call token (the second token in the pair). The statically assigned function symbol (the first token) token remains the same.

We assume that each textual subgoal in the program has a unique identifier. These take the form of letters[1] assigned statically to the program. For example, the program:

```
?- p(X).
p(A) :- q(A), r(A).
p(B) :- p(B), p(B).
q(C).
r(D).
```

will be assigned the following subgoal identifiers:

```
?- p_a(X).
p(A) :- q_b(A), r_c(A).
p(B) :- p_d(B), p_e(B).
q(C).
r(D).
```

We can now construct for any predicate call which occurs during the execution of a program a sequence of the subgoal call numbers of all its ancestors. If we use a dot '.' as a sequence constructor, the toplevel subgoal has call sequence a (with just one element), the first call to q/1 has call sequence $b.a$ and the second call to q/1 has call sequence $b.d.a$. We use these sequences as predicate call tokens. The function symbols and variables in the top level call require an empty sequence which we write as λ.

The unification procedure given in Section 6.2 is modified to:

[1] We have chosen letters to distinguish them from the numbers used to identify function symbols.

For each indexed substitution I in C,
  Replace all the variables in L and R with their values in I.
  Check corresponding subterms of resulting index terms
    If they have the same function symbol (ignoring indices)
      reindex all occurrences of the younger of the two symbols
      with the older.
      go to next subterm
    If one is *Var*
      replace all occurrences of it with the other side
      and go to next subterm
    Otherwise fail.
Gather resulting indexed substitutions into the output set.

That is instead of reindexing arbitrarily (that is, from left to right) the age of the symbols is taken into account. The age of a symbol is determined by the call sequence of its index; a symbol is older than another if its call sequence is shorter than the others. If the age is the same the choice is arbitrary.

7.4 Abstract Domain

An abstract substitution in the type inferencing domain (unless it is the $\perp$ element) consists of an indexed substitution mapping from local variables to indexed terms. Indexed terms are constructed from indexed function symbols and variables, supplemented with:

- Tagged choice terms (written OR).
- Top down loops (written TD).
- Bottom up loops (written BU).

We introduce our type language with some examples of abstract substitutions. This will illustrate the use of the new constructs and show the expressive power of the type language. In Section 7.5 we give a precise description of the relationship between an abstract domain based on the type language and the concrete domain.

7.4.1 Choice Terms

As a first example, consider the output of the following predicate[2]:

?- $p_a(X, Y)$.
p(A, B) :- $A=x_{1:\_}$, $B=y_{2:\_}$.
p(C, D) :- $C=z_{3:\_}$, $D=w_{4:\_}$.

[2]Note that we assume that indices for the function symbols and subgoals have been added to the program.

This returns the following set of concrete substitutions:

$\{\langle\ X/x_{1:a}\ Y/y_{2:a}\ \rangle$
$\langle\ X/z_{3:a}\ Y/w_{4:a}\ \rangle\ \}$

In the type language we represent this set as a single abstract substitution:

$\langle\ X/\text{OR}_a(x_{1:a}, z_{3:a})\ Y/\text{OR}_a(y_{2:a}, w_{4:a})\ \rangle$

This illustrates the use of tagged choice terms. The OR terms are tagged with the token identifying the call which created them—in the above, where there is only one call, this is *a*. The idea behind tags is that if a choice is made for one of these terms, then the corresponding choice must be made for all terms with the same tag[3]. This implies that all the OR terms with the same tag have the same number of arguments. Thus the above abstract substitution represents the set of concrete substitutions exactly. Without the tags—that is, with independent choice terms—the set represented would be:

$\{\langle\ X/x_{1:a}\ Y/y_{2:a}\ \rangle$
$\langle\ X/x_{1:a}\ Y/w_{4:a}\ \rangle$
$\langle\ X/z_{3:a}\ Y/y_{2:a}\ \rangle$
$\langle\ X/z_{3:a}\ Y/w_{4:a}\ \rangle\ \}$

The semantics for the abstract domain is defined in terms of a procedure for committing to one of the choices in an OR term. In the above, we can either commit to the first choice, where the resulting substitution is:

$\langle\ X/x_{1:a}\ Y/y_{2:a}\ \rangle$

or the second choice:

$\langle\ X/z_{3:a}\ Y/w_{4:a}\ \rangle$

7.4.2 Top-down Loops

The next example illustrates the top-down loop construct. Consider the output of the predicate:

?- $p_a(X)$.
$p(A)$:- $A=[]_{1:\_}$.
$p(B)$:- $B=[H|_{2:\_}T]$, $p_b(T)$.

This returns the set of concrete substitutions where X is bound to all lists of distinct variables:

[3]Similar constructs are used to represent alternatives in data-structures used in other fields; for example, feature terms [25].

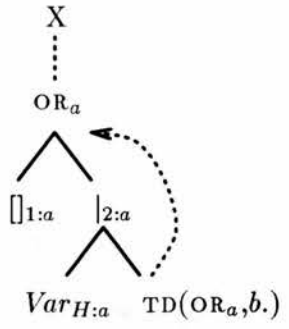


Figure 7.1: A graph of an abstract substitution representing the set of concrete substitutions where X is bound to a list of distinct variables formed in a top-down loop.

$\{ \langle X/[]_{1:a} \rangle$
$\langle X/[Var_{H:a} \; |_{2:a} \; []_{1:b.a}] \rangle$
$\langle X/[Var_{H:a} \; |_{2:a} \; [Var_{H:b.a} \; |_{2:b.a} \; []_{1:b.b.a}]] \rangle \; \ldots \; \}$

In each substitution, X is bound to a longer list with the function symbol index being incremented by a fixed amount. (Actually, it is the predicate call token which is being incremented). In the type language we represent this set as:

$\langle X/OR_a([]_{1:a}, [Var_{H:a} \; |_{2:a} \; TD(OR_a,b.)]) \rangle$

This abstract substitution is illustrated in Figure 7.1. It shows that X is bound to either $[]_{1:a}$ or a term whose principal functor is $|_{2:a}$ and which contains the loop term $TD(OR_a,b.)$ parameterized on the symbol which delimits the top of the loop and the increment for the indices.

Informally, the loop term represents some number of repetitions of the body of the loop (the part of the term between OR_a and $TD(OR_a,b.)$). The precise semantics of abstract substitutions are defined in terms of an unwinding function which is capable of committing to a choice which involves a loop and constructing a new term which represents the results. If we were to unwind the above loop once we get a term which represents all lists of different variables with a length of at least one:

$\langle X/[Var_{H:a} \; |_{2:a} \; OR_{b.a}([]_{1:b.a}, [Var_{H:b.a} \; |_{2:b.a} \; TD(OR_{b.a},b.)])] \rangle$

This is illustrated in Figure 7.2. Note that after unwinding, the indices in the loops have to be updated by adding the increment.

7.4.3 Bottom-up Loops

Another loop construct is used for bottom-up recursive data-structures, as constructed by the following predicate:

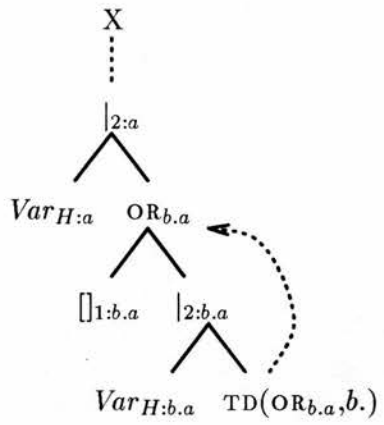


Figure 7.2: A graph of an abstract substitution representing the set of concrete substitutions where X is bound to a non-empty list of distinct variables formed in a top-down loop.

?- $Y=[]_{1:\_}$, $p_a(Y, X)$.
p(A, A).
p(B, C) :- $D=[H|_{2:\_}B]$, $p_b(D, C)$.

Similarly to the top-down case, this returns a set of substitutions where X is bound to every list of distinct variables:

$\{\langle X/[]_{1:\lambda}\ Y/[]_{1:\lambda}\rangle$
$\langle X/[Var_{H:a}\ |_{2:a}\ []_{1:\lambda}]\ Y/[]_{1:\lambda}\rangle$
$\langle X/[Var_{H:b.a}\ |_{2:b.a}\ [Var_{H:b.a}\ |_{2:b.a}\ []_{1:\lambda}]]\ Y/[]_{1:\lambda}\rangle\ \ldots\ \}$

We represent this set as:

$\langle X/OR_a([]_{1:\lambda}, [Var_{H:a}\ |_{2:a}\ BU(OR_a,b.)])\ \ Y/[]_{1:\lambda}\rangle$

The term bound to X is illustrated in Figure 7.3. Bottom-up loops unwind in the opposite direction from that of top-down loops. The unwound portion of the loop is added to the top of the non-looping choices in the term. For example, the above loop unwound once results in:

$\langle X/OR_{b.a}([Var_{H:a}\ |_{2:a}\ []_{1:\lambda}], [Var_{H:b.a}\ |_{2:b.a}\ BU(OR_{b.a},b.)])\ \ Y/[]_{1:\lambda}\rangle$

This is illustrated in Figure 7.4.

7.4.4 Combinations

Loops may be formed which contain other loops. For example, the following program:

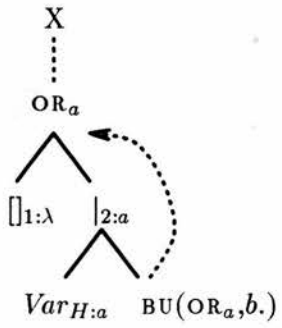


Figure 7.3: A graph of an abstract substitution representing the set of concrete substitutions where X is bound to a list of distinct variables formed in a bottom-up loop.

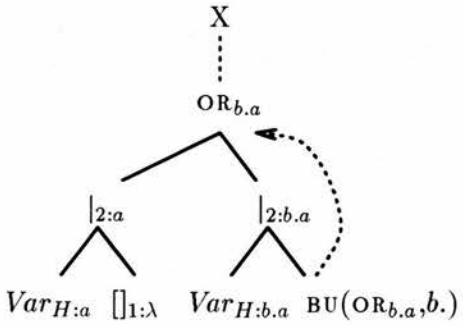


Figure 7.4: A graph of an abstract substitution representing the set of concrete substitutions where X is bound to a non-empty list of distinct variables formed in a bottom-up loop.

```
?- p_a(X).
p(A) :- A=[]_{1:_}.
p(B) :- B=[H1|_{2:_}T], q_b(H1), p_c(T).
q(C) :- C=[]_{3:_}.
q(D) :- D=[H|_{4:_}T], q_d(T).
```

will output lists each element of which is a list of variables:

$$\langle\ X/\mathrm{OR}_a([]_{1:a},\ [\mathrm{OR}_{b.a}([]_{3:b.a},\ [\mathrm{Var}_{H:b.a}\ |_{4:b.a}\ \mathrm{TD}(\mathrm{OR}_{b.a},d.)])\ |_{2:a}\ \mathrm{TD}(\mathrm{OR}_a,c.)]\ )\ \rangle$$

Unwinding the outer most loop results in:

$$\langle\ X/[\mathrm{OR}_{b.a}([]_{3:b.a},\ [\mathrm{Var}_{H:b.a}\ |_{4:b.a}\ \mathrm{TD}(\mathrm{OR}_{b.a},d.)])\ |_{2:a}\ \mathrm{OR}_{c.a}([]_{1:c.a},\ [\mathrm{OR}_{b.c.a}([]_{3:b.c.a},\ [\mathrm{Var}_{H:b.c.a}\ |_{4:b.c.a}\ \mathrm{TD}(\mathrm{OR}_{b.c.a},d.)])\ |_{2:c.a}\ \mathrm{TD}(\mathrm{OR}_{c.a},c.)]\ )\ ]\ \rangle$$

Note that the increment is inserted in the function symbol and variable indices at the point where the index is the same as that of the OR term at the top of the loop being unwound. Thus in the above the indices of the form *b.a* become *b.c.a* because *a* is the index of the OR term.

The following program constructs lists all of whose elements are the same:

```
?- p_a(X, Y).
p(A, B) :- A=[]_{1:_}.
p(C, D) :- C=[D|_{2:_}T], p_b(T, D).
```

This is represented by the abstract substitution:

$$\langle\ X/\mathrm{OR}_a([]_{1:a},\ [Var_{Y:\lambda}\ |_{2:a}\ \mathrm{TD}(\mathrm{OR}_a,b.)])\ \ Y/Var_{Y:\lambda}\ \rangle$$

To ensure that it is the same element which appears throughout the list, the unwinding operation does not update the indices of variables or function symbols which were formed outside the loop itself. This can be determined from their index; any index whose prefix is not the same as that of the OR term delimiting the loop is not incremented. The result of unwinding the above loop once is:

$$\langle\ X/[Var_{Y:\lambda}\ |_{2:a}\ \mathrm{OR}_{b.a}([]_{1:b.a},\ [Var_{Y:\lambda}\ |_{2:b.a}\ \mathrm{TD}(\mathrm{OR}_{b.a},b.)])]\ \ Y/Var_{Y:\lambda}\ \rangle$$

Here the variable $Var_{Y:\lambda}$ is not incremented because λ does not have a prefix *a*.

The next example is the output of the following program which relates a list to all its possible sublists:

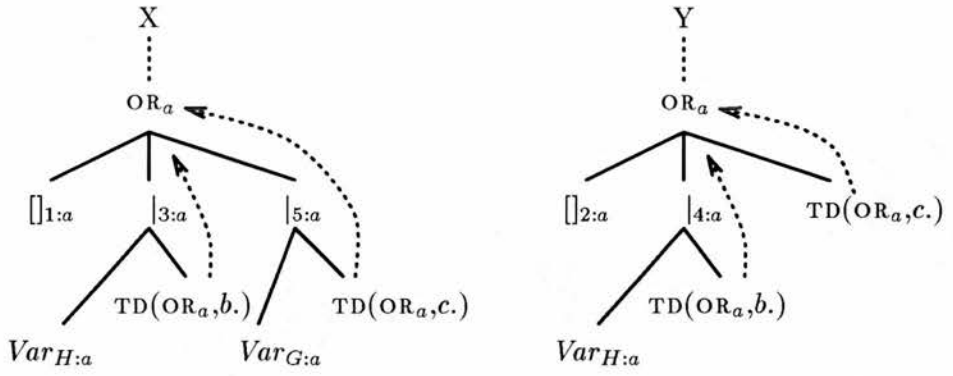


Figure 7.5: A graph illustrating the output of the sublist/2 predicate.

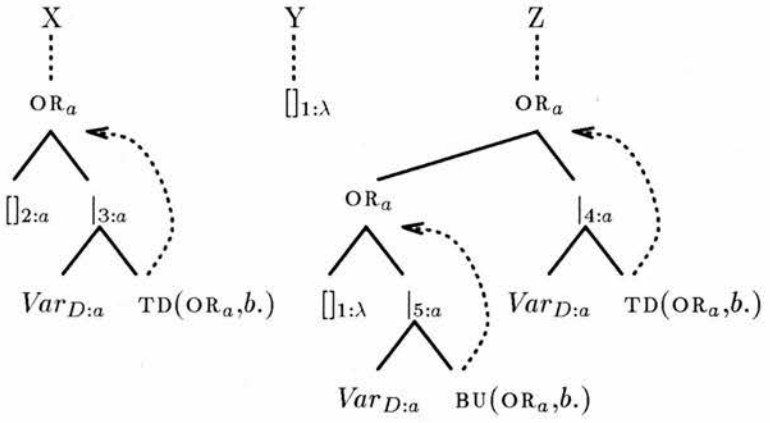


Figure 7.6: A graph illustrating the output of the palin/3 predicate.

?- sublist_a(X,Y).
sublist(A, B) :- A=$[]_{1:\_}$, B=$[]_{2:\_}$.
sublist(C, D) :- C=[H$|_{3:_}$E], D=[H$|_{4:_}$F], sublist_b(E,F).
sublist(I, J) :- I=[G$|_{5:_}$K], sublist_c(K, J).

The program results in the substitution:

$\langle$ X/$OR_a([]_{1:a}$, [$Var_{H:a}$ $|_{3:a}$ $TD(OR_a,b.)$], [$Var_{G:a}$ $|_{5:a}$ $TD(OR_a,c.)$])
Y/$OR_a([]_{2:a}$, [$Var_{H:a}$ $|_{4:a}$ $TD(OR_a,b.)$], $TD(OR_a,c.)$) $\rangle$

This is illustrated in Figure 7.5. This substitution shows the use of tagged OR terms to represent relationships between data-structures. If we unwind the above loop choosing the second of the three possible choices, then the same element $Var_{H:a}$ is added to both X and Y. If the third choice is taken an element is added to X only because the Y's third branch is an empty loop. Thus we can infer that Y is a sublist of X because some elements in X may be missed out from Y, but with the same relative order.

Top-down and bottom-up loops can also occur together. Consider a call to the palin/3 program:

```
?- Y=[]1:_, palin_a(X, Y, Z).
palin(A, B, C) :- A=[]2:_, B=C.
palin(E, F, G) :- E=[D|3:_H], G=[D|4:_I],
   J=[D|5:_F], palin_b(H, J, I).
```

which results in the following abstract substitution:

$$\begin{array}{l}\langle\ \mathrm{X}/\mathrm{OR}_a([]_{2:a},\ [\mathit{Var}_{D:a}\ |_{3:a}\ \mathrm{TD}(\mathrm{OR}_a,b.)]) \\ \ \ \mathrm{Y}/[]_{1:\lambda} \\ \ \ \mathrm{Z}/\mathrm{OR}_a(\ \mathrm{OR}_a([]_{1:\lambda},[\mathit{Var}_{D:a}\ |_{5:a}\ \mathrm{BU}(\mathrm{OR}_a,b.)]), \\ \qquad [\mathit{Var}_{D:a}\ |_{4:a}\ \mathrm{TD}(\mathrm{OR}_a,b.)])\ \rangle\end{array}$$

This shows two loops sharing the same choice tag (and therefore having the same number of repetitions) but where one is in the nonlooping branch of the other. This is illustrated in Figure 7.6. When this loop is unwound once it results in the substitution:

$$\begin{array}{l}\langle\ \mathrm{X}/[\mathit{Var}_{D:a}\ |_{3:a}\ \mathrm{OR}_{b.a}([]_{2:b.a},\ [\mathit{Var}_{D:b.a}\ |_{3:b.a}\ \mathrm{TD}(\mathrm{OR}_{b.a},b.)])] \\ \ \ \mathrm{Y}/[]_{1:\lambda} \\ \ \ \mathrm{Z}/[\mathit{Var}_{D:a}\ |_{4:a} \\ \qquad \mathrm{OR}_{b.a}(\ \mathrm{OR}_{b.a}([\mathit{Var}_{D:b.a}\ |_{5:b.a}\ []_{1:\lambda}],[\mathit{Var}_{D:b.a}\ |_{5:b.a}\ \mathrm{BU}(\mathrm{OR}_{b.a},b.)]), \\ \qquad\quad [\mathit{Var}_{D:b.a}\ |_{4:b.a}\ \mathrm{TD}(\mathrm{OR}_{b.a},b.)])\ ]\ \rangle\end{array}$$

With each unwinding of the loop a new variable is added to the front and the back halves of Z's list. These additions occur from the centre of the list. This structure accurately represents a palindromic list with even length.

7.5 Concretization

We now define more precisely the relationship between the concrete and the abstract domains by providing the concretization *conc* function introduced in Chapter 4.

The input to *conc* is an abstract substitution A and it returns a set of indexed substitutions. The procedure is:

```
If A is ⊥ return the empty set
else
   return the set formed by generating all possible
   indexed substitutions from A.
```

An indexed substitution can be generated by starting from the local variables and traversing down the terms, repeatedly picking an OR term from A and making a choice, until there are no OR terms in the substitution. A concrete indexed substitution results when there are no more OR terms nor loop terms in the substitution.

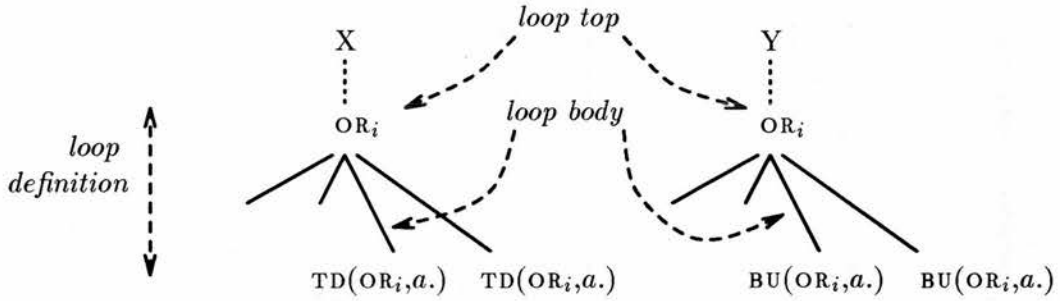


Figure 7.7: Loop Skeleton.

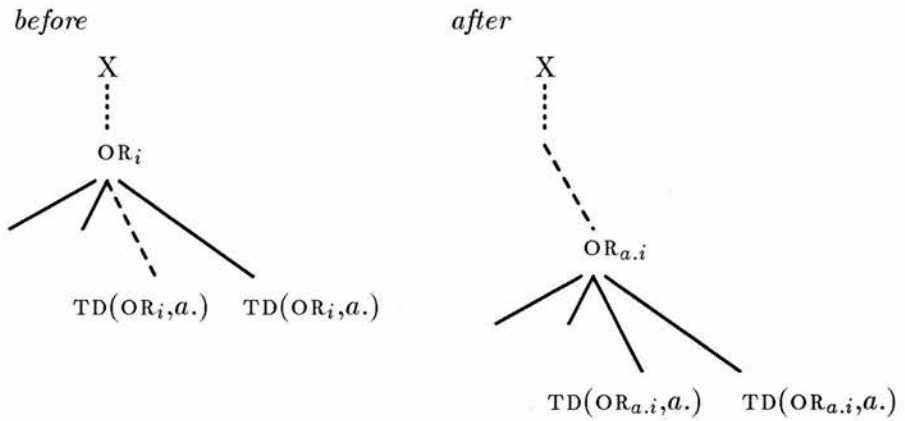


Figure 7.8: Skeleton of unwinding a top-down loop.

If, however, the OR branch in question forms a loop it is necessary to unwind the loop as well. Consider the situation illustrated in Figure 7.7. This shows the skeleton of a substitution just before unwinding the third of the OR$_i$ choices. In this situation, we have a number of OR$_i$ terms and we have just picked one of the choices that contains a loop term pointing back to the OR$_i$ term. We refer to the OR$_i$ functors as the loop tops, the terms below the loop tops as the loop definition and the part of the term between the loop top and the TD(OR$_i$,a.) (or BU(OR$_i$,a.)) term as the loop body. Note that the loop body depends on the particular choice made. The second argument to TD(OR$_i$,a.), that is a., is called the increment.

A loop is unwound by instantiating one iteration of the loop. Because the OR$_i$ choices are tagged all of these loops are unwound simultaneously which means that the variables throughout the loop definition are reindexed consistently. However the procedures for unwinding individual loops depend on whether the loop is top-down or bottom-up. We describe these procedures separately below.

The skeleton of a top-down loop before and after unwinding is given in Figure 7.8

where the loop body is marked as a dashed line. The procedure for unwinding a top-down loop is as follows:

1. Make a copy of the loop body, the term in the branch of the OR_i term being unwound. We illustrate unwinding a top-down loop using the list of variables example substitution used earlier:

   $\langle\ \text{X}/\text{OR}_a([]_{1:a}, [Var_{H:a}\ |_{2:a}\ \text{TD}(\text{OR}_a,b.)])\ \rangle$

   The loop body is the term:

   $[Var_{H:a}\ |_{2:a}\ \text{TD}(\text{OR}_a,b.)]$

2. Make a copy of the whole loop. Reindex the variables and function symbols and retag the OR_i terms in the copy by applying the increment in the TD term. Reindexing inserts the loop increment into all function symbol and variable indices and the OR tags which have i as a prefix. In the example, the loop is the term:

   $\text{OR}_a([]_{1:a}, [Var_{H:a}\ |_{2:a}\ \text{TD}(\text{OR}_a,b.)])$

   and when reindexed/retagged it becomes:

   $\text{OR}_{b.a}([]_{1:b.a}, [Var_{H:b.a}\ |_{2:b.a}\ \text{TD}(\text{OR}_{b.a},b.)])$

   where the increment is $b.$ and the prefix is $a.$

3. In the original substitution, replace the loop definition by the copy of the loop body. This gives:

   $\langle\ \text{X}/[Var_{H:a}\ |_{2:a}\ \text{TD}(\text{OR}_a,b.)]\ \rangle$

4. Finally, replace the TD term with the reindexed loop term. This gives:

   $\langle\ \text{X}/[Var_{H:a}\ |_{2:a}\ \text{OR}_{b.a}([]_{1:b.a}, [Var_{H:b.a}\ |_{2:b.a}\ \text{TD}(\text{OR}_{b.a},b.)])]\ \rangle$

The above assumes that there is only one loop term in the body. If there is more than one, each with a different increment, it is necessary to reindex a number of copies of the loop definition. For example, the output of the program:

?- $\text{p}_a(\text{X})$.
p(A) :- A=$\text{leaf}_{1:\_}$.
p(B) :- B=$\text{tree}_{2:\_}$(E, L, R), p_b(L), p_c(R).

is:

$\langle \text{X}/\text{OR}_a(\text{leaf}_{1:a}, \text{tree}_{2:a}(\ Var_{E:a}, \text{TD}(\text{OR}_a,b.), \text{TD}(\text{OR}_a,c.))))\rangle$.

After unwinding once, this becomes:

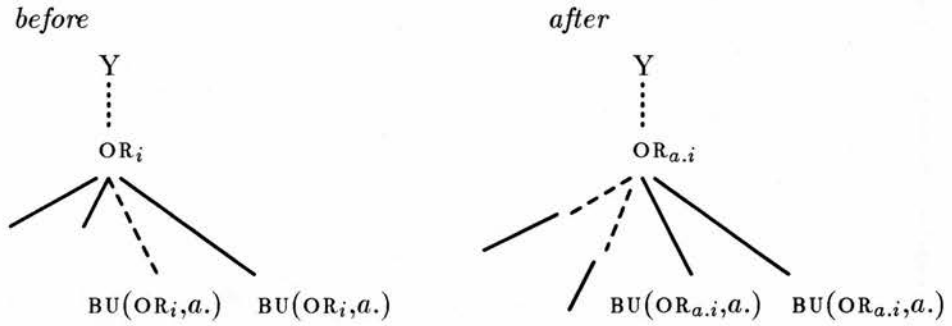


Figure 7.9: Skeleton of unwinding a bottom-up loop.

$\langle$X/tree$_{2:a}$($Var_{E:a}$,
OR$_{b.a}$(leaf$_{1:b.a}$,tree$_{2:b.a}$($Var_{E:b.a}$,TD(OR$_{b.a}$,b.), TD(OR$_{b.a}$,c.)))
OR$_{c.a}$(leaf$_{1:c.a}$,tree$_{2:c.a}$($Var_{E:c.a}$,TD(OR$_{c.a}$,b.), TD(OR$_{c.a}$,c.)))))$\rangle$.

The procedure for unwinding a bottom-up loop is as follows (see Figure 7.9):

1. Make a copy of the loop body. We illustrate unwinding bottom-up loops using the list of variables formed by a bottom-up recursion:

   $\langle$ X/OR$_a$([]$_{1:\lambda}$, [$Var_{H:a}$ |$_{2:a}$ BU(OR$_a$,b.)]) Y/[]$_{1:\lambda}\rangle$

   The loop body is:

   [$Var_{H:a}$ |$_{2:a}$ BU(OR$_a$,b.)]

2. In the original substitution, reindex the variables and function symbols and retag the OR$_i$ terms in the copy by applying the increment in the BU term. In the example, this gives:

   $\langle$ X/OR$_{b.a}$([]$_{1:\lambda}$, [$Var_{H:b.a}$ |$_{2:b.a}$ BU(OR$_{b.a}$,b.)]) Y/[]$_{1:\lambda}\rangle$

3. In the substitution, make a copy of all the non-looping branches of the OR$_i$ term. Here this is the single term:

   []$_{1:\lambda}$

4. Replace all non-looping branches with a copy of the loop body. This gives:

   $\langle$ X/OR$_{b.a}$([$Var_{H:a}$ |$_{2:a}$ BU(OR$_a$,b.)],[$Var_{H:b.a}$ |$_{2:b.a}$ BU(OR$_{b.a}$,b.)])
   Y/[]$_{1:\lambda}\rangle$

   and replace the BU with the copy of the term that was originally in each branch. The result is:

$\langle$ X/$\text{OR}_{b.a}$([$Var_{H:a}$ $|_{2:a}$ $[]_{1:\lambda}$],[$Var_{H:b.a}$ $|_{2:b.a}$ BU($\text{OR}_{b.a}$,*b.*)])
Y/$[]_{1:\lambda}\rangle$

7.6 Termination

As we have described in Chapters 4 and 5, in order to guarantee that an abstract interpreter will terminate we need to restrict the abstract domain so that there are no indefinitely increasing chains in the subsumption ordering over its abstract domain. In the domain we have described above this is *not* the case, because it contains standard substitutions as a subset.

The solution we propose for this problem is to ensure that any chain which a Prolog program can actually construct will be recognized and the top element of the chain will be generalized to immediately. That is, the abstract domain in general does not meet our criteria but the interpreter may be arranged so that in practice it terminate.

A chain in the above is a sequence of abstract substitutions which represent increasing sets of concrete substitutions. The important property which programs have is that any chain is 'rational' in the sense that chains increase by a fixed amount for each iteration. That is, the chain has a fixed periodicity. This is because only recursive programs can construct indefinitely increasing chains[4] and a recursive program involves repeated execution of the same piece of code which has the same effects with each iteration.

The abstract domain is capable of representing repetitive data-structures using the loop constructs (that is, TD and BU). These allow us to construct the top element in any chain of fixed periodicity. Thus, provided it is possible to detect chains and extract the effects of one iteration, we can always construct the top element and reach a fix point. In our framework (see Chapter 5) we allowed for primitives in the abstract interpreter which are capable of generalizing any abstract substitution.

We now develop the rule for recognizing chains using a simple example program which constructs a top-down loop:

?- p_a(X).
p(A) :- A=$[]_{1:\_}$.
p(B) :- B=$[H|_{2:\_}T]$, p_b(T).

The successive substitutions on the exit stream of the extension table entry for the call p(X) with no generalization check are:

[4]Recall that we are considering only pure Prolog. Here programs are a finite collection of clauses constructed from a finite collection of function symbols. In particular, we are not handling numbers.

$\perp$,
$\langle$ X/$[]_{1:a}$ $\rangle$
$\langle$ X/$\text{OR}_a([]_{1:a}, [Var_{H:a} \mid_{2:a} []_{1:b.a}])$ $\rangle$
$\langle$ X/$\text{OR}_a([]_{1:a}, [Var_{H:a} \mid_{2:a} \text{OR}_{b.a}([]_{1:b.a}, [Var_{H:b.a} \mid_{2:b.a} []_{1:b.b.a}])])$ $\rangle$
$\vdots$

We can recognize that the same piece of code has been executed by comparing the second and third substitutions. The term $[]_{1:b.a}$ in the third substitution is a reindexing of the entire term bound to X in the second substitution. Since the two terms have the same symbol token (that is, the left hand side of the ':' pair making up the index), namely 1, we know that these terms were constructed in the same clause[5]. This allows us to introduce a top-down loop in place of $[]_{1:b.a}$. The increment of the loop is the difference between the indices of the two terms; that is, b. The third element can then be generalized to:

$\langle$ X/$\text{OR}_a([]_{1:a}, [Var_{H:a} \mid_{2:a} \text{TD}(\text{OR}_a, b.)])$ $\rangle$

This element subsumes all subsequent elements allowing the fix point iteration to halt.

The general rule used for generalizing chains is:

> For any variable V in two exit substitutions
> if V's value in the later exit contains a reindexing
> of its value in the earlier exit
> then generalize to a top-down loop by replacing this
> renamed term with a TD term whose increment
> is the difference between the indices of the two values.

The above scheme works reasonably well within our framework for abstract interpretation (see Chapter 5) because the successive elements in the chain of predicate exits from a call to a recursive predicate are conveniently available in the corresponding exit stream for the call in the extension table.

However, the problem of recognising chains and generalizing them is in general undecidable and we suggest only that it is possible to perform such generalizations for particular classes of programs.

To terminate computations building bottom-up recursive data-structures it is necessary to compare successive predicate *calls* instead of exits. Consider the following predicate:

```
?- p_a(X).
p(A) :- B=f1(A), p_b(B).
```

The substitutions for the successive calls are:

[5] We assume that textual variable names are chosen so that they are unique to the entire predicate. This allows us to detect variables created from the same textual variable.

$\langle$ X/$Var_{X:\lambda}$ $\rangle$
$\langle$ B/$f_{1:a}(Var_{X:\lambda})$ $\rangle$
$\langle$ B/$f_{1:b.a}(f_{1:a}(Var_{X:\lambda}))$ $\rangle$

It is possible to use a similar rule to the top-down case above to generalize the second call to:

$\langle$ B/$\text{OR}_a(Var_{X:\lambda}, f_{1:a}(\text{BU}(\text{OR}_a,b.)))$ $\rangle$

However, such a scheme does not fit into our abstract interpretation framework very well. It is necessary to examine the entire extension table looking for an earlier call to the same predicate whose exit stream is not yet closed (and which therefore the current call is recursing on). Even though this generalized call will terminate, this is not ideal since it will not return the most elegant result and it is difficult to ensure that any top-down loops which are formed in the same predicate are in step with the bottom-up loops.

To handle bottom-up recursions properly it is necessary to *replace* the original call with the generalized call and restart the computation. This would require quite extensive changes to our abstract interpretation framework.

7.7 Body Unification

Here we sketch the procedure for abstract body unification and give some examples. Since they are very much alike, body unification in the abstract domain is based on body unification for the concrete domain, with the following additional conditions needed to handle OR_i choices and loops:

- Two OR choices are unified by attempting to unify every possible choice from either side and joining the results by creating a new tag for the OR choices.

  For example, if we unify X with Y where[6]:

  $\langle$X/OR_a(a, b, c)
  Y/OR_b(b, c, d)
  Z/OR_b(e, f, g) $\rangle$

  This results in:

  $\langle$X/OR_a(b, c)
  Y/OR_a(b, c)
  Z/OR_a(e, f) $\rangle$

  That is every choice from X has been unified with every choice from Y. The only successful combinations are the second choice of X with the first from Y, and the

[6] Dropping the function symbols indices

third choice from X with the second for Y. Note that the effects of committing to the choices is reflected in the rest of the substitution; in the above the OR term bound to Z has been updated.

- A loop is unified with a term by first unwinding the loop sufficiently so that the loop definition is never actually unified against a non-loop.

  For example, if we unify X and [A|B] where:

  $\langle \mathrm{X}/\mathrm{OR}_a([]_{1:a}, [Var_{H:a}|_{2:a}\mathrm{TD}(\mathrm{OR}_a,b.)])$
  $\mathrm{A}/Var_{A:\lambda}$
  $\mathrm{B}/Var_{B:\lambda}\ \rangle$

  This results in:

  $\langle\ \mathrm{X}/[Var_{H:a}\ |_{2:a}\ \mathrm{OR}_{b.a}([]_{1:b.a}, [Var_{H:b.a}|_{2:b.a}\mathrm{TD}(\mathrm{OR}_{b.a},b.)])]$
  $\mathrm{A}/Var_{H:a}$
  $\mathrm{B}/\mathrm{OR}_{b.a}([]_{1:b.a}, [Var_{H:b.a}|_{2:b.a}\mathrm{TD}(\mathrm{OR}_{b.a},b.)])\ \rangle$

  Here, X's loop has been unwound so that A and B are not unified with terms within the loop repetition. This would have the effect of binding A to all the variables in the list, rather than just the first one.

- Two loops are unified without unwinding except if a TD or BU term is reached on just one side. In this case the loop should be expanded to increase it's periodicity and the entire unification restarted.

  For example, if we unify X and A where:

  $\langle \mathrm{X}/\mathrm{OR}_a([]_{1:a}, [Var_{H:a}|_{2:a}\mathrm{TD}(\mathrm{OR}_a,b.)])$
  $\mathrm{A}/\mathrm{OR}_c([]_{3:c}, [\mathrm{a}_{4:c},\mathrm{b}_{5:c}|_{6:c}\mathrm{TD}(\mathrm{OR}_c,d.)])\ \rangle$

  This results in:

  $\langle\ \mathrm{X}/\mathrm{OR}_c([]_{3:c}, [\mathrm{a}_{4:c},\mathrm{b}_{5:c}|_{6:c}\mathrm{TD}(\mathrm{OR}_c,d.)])$
  $\mathrm{A}/\mathrm{OR}_c([]_{3:c}, [\mathrm{a}_{4:c},\mathrm{b}_{5:c}|_{6:c}\mathrm{TD}(\mathrm{OR}_c,d.)])\ \rangle$

  Since A is a loop with a periodicity of 2, it is necessary to expand X's loop so that it has the same periodicity; that is, X's value should become:

  $\mathrm{OR}_a([]_{1:a}, [Var_{H:a}, Var_{H:b.a}|_{2:b.a}\mathrm{TD}(\mathrm{OR}_a,b.)])$

  This expansion occurs during the unification of the two loops when an attempt is made to unify a loop term (that is, either TD or BU) with a standard term. It is then necessary to restart the unification of the loops.

7.8 Implementation and Comparison

The type language we have sketched above allows the relationships between data-structures to be represented, mainly by the use of tagged OR choices. The information inferred subsumes that inferred by many other abstract domains. Thus an abstract domain based on this type language could be regarded as a general abstract domain, forming the basis of many applications.

The language allows us to describe techniques such as those used in qsort/3. However, our abstract interpreter based on this type language is not complete, and therefore we cannot actually detect this class of techniques for all programs. We have, however, implemented an abstract interpreter which works for the top-down example programs we have given.

Relationships between data-structures allow us to reason about the relative lengths of recursive data-structures. This may allow compilers to make extra optimizations. For example, knowing that a data-structure is of constant length may enable it to be represented as a contiguous series of memory cells rather than as a linked list.

The closest piece of previous work to our proposal is the type inferencing scheme of Mulkers *et al* [55] (see Section 4.3.2). Here data-structures are represented as trees, with back arcs representing loops, whose nodes are labelled with functions symbols and a single token representing all variables. An additional component in the abstract substitution represents sharing between different parts of the trees. This corresponds to our use of indices on function symbols and variables. In Mulkers' work relationships between data-structures cannot be represented and so the language is not suitable for our techniques description application.

Mulkers' scheme ensures termination by detecting repeated function symbols along branches of the type trees and then generalizing to a loop construct. Our scheme for ensuring the termination of our abstract interpreter examines successive calls and exits. Our termination strategy is conservative in that it is reluctant to generalize to a loop construct until there is firm evidence that a recursion has been entered; namely, that a clause has been executed more than once. This allows data-structures to be more accurately inferred. For example, the output of the program:

?- $p_a(X)$.
p(A) :- $A=[]_{1:\_}$.
p(B) :- $B=[H1|_{2:\_}T]$, $q_b(H1)$, $p_c(T)$.
q(C) :- $C=[]_{3:\_}$.
q(D) :- $D=[H|_{4:\_}T]$, $q_d(T)$.

is accurately represented as a list of lists rather than a nested list as would be inferred by Mulkers' scheme.

A similar scheme for type inference is described by Bansal and Sterling [4] (see Sec-

tion 4.3.2). Their type language represents variables sharing between data-structures but not relationships between recursive data-structures. They assume that the programs which they analyse always terminate.

7.9 Summary

In this chapter, we have

- sketched an abstract domain to infer a class of type information suitable for defining certain programming techniques.
- used tagged choice terms in the domain to accurately represent alternative values of variables.
- used loop constructs in the domain to represent recursive data-structures with a fixed periodicity.
- used a termination criterion based on detecting repeated clause execution, rather than repeated function symbols in data-structures.

Chapter 8

Summary, Contributions and Further Work

8.1 Introduction

The background of the research we have presented is the problem of enhancing Looi's P-frame language used in APROPOS2. This required us to develop a techniques description language, and to develop abstract interpretation based methods for inferring information sufficient for detecting instances of techniques in student's code. In doing this we have made a number of contributions to the areas of Prolog programming techniques and abstract interpretation.

8.2 Summary

The following summarizes each chapter of this thesis:

- In Chapter 1, we described the workings of APROPOS2, showed some of its limitations and suggested an approach for overcoming those limitations.
- In Chapter 2, we reviewed some recent work on Prolog programming techniques. Our conclusions were that none of this work is suitable as the basis for a more expressive P-frame language, primarily because the techniques are described in a largely syntactic way and thus difficult to detect in code. This motivated our own design for a techniques description language.
- In Chapter 3, we developed our own language for describing techniques. The main feature used by this language for describing techniques is the effects of unifications which occur during execution. These effects are expressed in terms of two relations: inclusion and sharing. We demonstrated how our techniques language might be used to enhance APROPOS2, and showed some of the limitations

of our approach.

- In Chapter 4, we reviewed work on abstract interpretation. We gave a brief presentation of the theory, described some recent applications of abstract interpretation to Prolog and described the different kinds of semantics used as a basis for abstract interpreters.

- In Chapter 5, we developed a framework for Prolog abstract interpretation. This framework defines properties that abstract domains must satisfy to guarantee that their analysis is both correct and terminating. We also presented a complete implementation of the framework in Prolog. This needs only to be instantiated by suitable definitions of primitives to yield an executable abstract interpreter. Our framework is based on that of Wærn [64], but it is an improvement in that it can often iterate to a fixpoint in fewer steps.

- In Chapter 6, we developed a simple abstract domain for inferring inclusion and sharing. This domain is similar to abstract domains developed for detecting shared data-structures, but is more precise in certain circumstances. Tools based upon this abstract domain allow programming techniques which are defined using inclusion and sharing to be detected. We also described detailed procedures for the primitives needed for instantiating our abstract interpretation framework.

- In Chapter 7, we sketched a more complex abstract domain capable of representing good quality type information. This enabled a broader range of programming techniques to be accurately specified and thus overcomes some of the limitations of our techniques description language. It may also have other uses in, for example, optimizing code generated by Prolog compilers. We suggested ways of ensuring that an abstract interpreter based on this domain can be made to terminate.

8.3 Contributions

We believe that we have made the following contributions:

8.3.1 Techniques Description Language

We have designed a powerful language for describing a range of Prolog techniques, and have used it to precisely define techniques such as accumulator pairs and difference structures. Previously, such techniques have only been defined informally using simple syntactic schemata, or using example programs. Our language is based on dynamic features, in particular on the effects of unifications which occur during execution. These effects are expressed in terms of two relations: inclusion and sharing, which are defined over a particular equality over indexed terms.

8.3.2 A Framework for Prolog Abstract Interpretation

We have developed a framework for Prolog abstract interpretation based on OLDT resolution. The framework is described in Prolog itself leading directly to an implementation. Our framework is based on that of Wærn [64], but it is an improvement in that it can often iterate to a fixpoint in fewer steps.

8.3.3 An Abstract Domain for Inferring Inclusion and Sharing

We have developed and implemented an abstract domain which infers inclusion and sharing information. This is similar to, but more precise than, the abstract domain described by Bruynooghe [9] to infer information for detecting shared data-structures in Prolog. It is based in an instrumented concrete description of Prolog which implements indexed term equality. The representation used in the abstract domain is similar to that of Jacobs and Langen [34] though with quite different semantics.

8.3.4 An Abstract Domain for Inferring Type Information

We have sketched the design of an abstract domain which can represent good quality type information; in particular, it can represent relationships between data-structures such as 'list a is the same length as list b'. We are not aware of any other type language in which it is possible to express and infer such relationships. The abstract domain is very complex and it requires a novel approach to ensuring termination.

8.4 Further Work

We have not actually modified APROPOS2 to judge whether the enhancements we suggest are practical. This would form an obvious extension to our work, but it would involve reimplementing large parts of APROPOS2. Time constraints do not allow us to do justice to this task.

We also suggest the following areas for further work:

8.4.1 Extended Techniques Description Language

The language we have developed for describing techniques covers only some techniques; in particular, those which manipulate data-structures using unification. This can be extended in a number of ways:

- Even simple Prolog programs make extensive use of numbers. Number equivalents to the *Up*, *Down* and *Acc*/*AccResult* techniques need to be provided. For example, the following predicate length/3 returns the length of a list:

```
length([], N, N).
length([_|T], N0, N) :-
    N1 is N0+1,
    length(T, N1, N).
?- length([a,b,c], 0, N).
```

The second and third arguments in the second clause form an instance of the general accumulator pair technique, but one which lies outside our description because the variables are not related by inclusion. We could, however, treat the call to the is/2 built in as forming an inclusion-like relation between the variables N1 and N0. Note however that there is no number equivalent to the difference structure techniques.

- Many techniques may be seen as specifying control flow patterns. Many of these, for example those of Gegg-Harrison [32] (see Chapter 2), may be accurately specified using a predicate level technique specification. However, more complex forms of Prolog control flow, such as those which involve non-determinate calls, would require an extended language. To illustrate, consider a description of the standard member/2 predicate:

```
member(X, [X|_]).
member(X, [_|Y]) :- member(X, Y).
?- member(X, [a,b,c]).
```

An important part of this description is that the call is non-determinate and capable in general of succeeding a number of times. Contrast this to the similar predicate member_chk/2 which uses a cut to force determinacy:

```
member_chk(X, [X|_]) :- !.
member_chk(X, [_|Y]) :- member_chk(X, Y).
?- member_chk(X, [a,b,c]).
```

It is not possible to express this distinction in our techniques description language.

8.4.2 Improved Inclusion and Sharing Domain

We believe that the precision of our inclusion and sharing abstract domain may be improved by allowing type annotations to the term descriptors in an abstract substitution. A similar idea was used by Bruynooghe [9]. This domain would be subsumed by our type inferencing domain, but would be somewhat simpler.

It is also possible to define a very simple abstract domain for inferring inclusion only. In such a domain an abstract substitution would consist of the inclusion relation itself; that is, a set of ordered pairs of the local variables. This domain would allow a

subset of our techniques to be detected, but would be relatively cheap to compute. For example, unification may be defined as follows:

- To unify a variable X with a term containing variables $Y_1 \ldots Y_n$, add the pairs X-Y_1 ...X-Y_n to the abstract substitution, together with sufficient extra pairs to maintain the transitivity of the set.
- To unify a local variable X with another local variable Y add the pairs X-Y and Y-X to the abstract substitution, together with sufficient extra pairs to maintain the transitivity of the set.

It may also be possible to make use of the information inferred by the inclusion and sharing abstract domain in other applications, in particular, in optimizing compilers. This would require the abstract domain to be based on a different instrumented concrete description of Prolog which reflected a real Prolog implementation, but the main ideas would be directly applicable.

8.4.3 Develop the Type Inference Domain

We have only sketched the abstract domain for inferring types. It still requires a suitable abstract interpreter to ensure that the analyses produce reasonable output.

It is also necessary to investigate which kind of extra techniques can be precisely defined using our extended type information.

It may also be possible to make use of the information inferred by the type abstract domain in other applications, in particular, optimizing compilers. Again this needs to be based on a different instrumented concrete description of Prolog.

Bibliography

[1] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[2] B. Adelson. Problem solving and the development of abstract categories in programming languages. *Memory and Cognition*, 9(4):422–433, 1981.

[3] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 1–15, 1986.

[4] A. Bansal and L. Sterling. An abstract interpretation scheme for logic programs based on type expressions. *New Generation Computing*, 7:273–324, 1990.

[5] D. Barker-Plummer. Cliche programming in Prolog. In M. Bruynooghe, editor, *Proceedings of the Second Workshop on Meta-Programming in Logic*, pages 247–256, Leuven, Belgium, 1990.

[6] P. Brna, A. Bundy, T. Dodd, M. Eisenstadt, C.K. Looi, H. Pain, D. Robertson, B. Smith, and M. van Someren. Prolog programming techniques. *Instructional Science*, 19(4/5), 1990.

[7] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. A revised version of KU Leuven CW report 62. To appear in Journal of Logic Programming, 1988.

[8] M. Bruynooghe and G. Janssens. An instance of abstract interpretation integrating type and mode inferencing. In *Fifth Symposium and Conference on Logic Programming*, pages 669–683, Seattle, Washington, 1988. MIT Press.

[9] M. Bruynooghe, G. Janssens, A. Callebaut, and B. Demoen. Abstract interpretation: Towards the global optimization of Prolog programs. In *Logic Programming Symposium*, pages 192–204, San Francisco, California, 1987. IEEE.

[10] F. Bry. Query evaluation in recursive databases: bottom-up and top-down reconciled. *Data and Knowledge Engineering Review*, 5(4):289–312, 1990.

[11] A. Bundy. Proposal for a recursive techniques editor for Prolog. Research Report 394, Department of Artificial Intelligence, University of Edinburgh, 1988.

[12] A. Bundy, G. Grosse, and P. Brna. A recursive techniques editor for Prolog. *Instructional Science*, 19(4/5), 1990.

[13] M. Codish, D. Dams, and E. Yardeni. Bottom-up abstract interpretation of logic programs. Unpublished manuscript available from authors, 1990.

[14] M. Codish, J. Gallagher, and E. Shapiro. Using safe approximations of fixed points for analysis of logic programs. In H. Abramson and M. Rogers, editors, *Proceedings of the First Workshop on Meta-Programming in Logic-Programming*, pages 233–261, Bristol, England, 1988. MIT Press.

[15] M. Corsini and G. Filè. The abstract interpretation of logic programs: A general algorithm and its correctness. Research Report, Dept. of Pure and Applied Mathematics, University of Padova, Italy, 1988.

[16] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[17] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E. J. Neuhold, editor, *Formal Descriptions of Programming Concepts*, pages 237–277. North Holland, 1978.

[18] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 269–282, 1979.

[19] S. Debray. Efficient dataflow analysis of logic programs. Unpublished manuscript available from author, 1990.

[20] S. Debray and R. Ramakrishnan. Canonical computations of logic programs. Unpublished manuscript available from authors, 1990.

[21] S. Debray and D. S. Warren. Automatic mode inference for Prolog programs. In *Logic Programming Symposium*, pages 78–88, Salt Lake City, Utah, 1986. IEEE.

[22] S. Debray and D. S. Warren. Automatic mode inference for logic programs. *Journal of Logic Programming*, 5(3):207–230, 1988.

[23] D. DeGroot. Restricted And-parallelism. In H. Aiso, editor, *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 471–478, Tokyo, Japan, 1984. North Holland.

[24] S. W. Dietrich. Extension tables: memo relations in logic programming. In *Logic Programming Symposium*, pages 264–272, San Francisco, California, 1987. IEEE.

[25] J. Dörre and A. Eisele. Determining consistency of feature terms with distributed disjunctions. In D. Metzing, editor, *GWAI-89 13th German Workshop on Artificial Intelligence*, pages 270–279. Springer Verlag, 1989.

[26] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of operational behaviour of logic languages. *Theoretical Computer Science*, 69:289–318, 1989. Also University of Pisa, TR 10/88.

[27] M. Fitting. A Kripke-Kleene semantics for logic programming. *Journal of Logic Programming*, 2(4):295–312, 1985.

[28] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialization. In D. H. D. Warren and P. Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 732–746, Jerusalem, Israel, 1990. MIT Press.

[29] J. Gallagher and M. Bruynooghe. Some low-level source transformations for logic programs. In M. Bruynooghe, editor, *Proceedings of the Second Workshop on Meta-Programming in Logic*, pages 229–244, Leuven, Belgium, 1990.

[30] J. Gallagher, M. Codish, and E. Shapiro. Specialization of Prolog and FCP programs using abstract interpretation. *New Generation Computing*, 6(2/3):159–186, 1988.

[31] H. Gallaire, J. Minker, and J. Nicolas. Logic and databases: a deductive approach. *Computing Surveys*, 16(1):154–185, 1984.

[32] T.S. Gegg-Harrison. Learning Prolog in a schema-based environment. *Instructional Science*, 19(4/5), 1990.

[33] R. Harper, D. MacQueen, and R. Milner. Standard ML. Technical Report ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, 1986.

[34] D. Jacobs and A. Langen. Accurate and efficient approximation of variable aliasing in logic programs. In E. Lusk and R. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 154–165, Cleveland, Ohio, 1989. MIT Press.

[35] W. L. Johnson. *Intention-Based Diagnosis of Errors in Novice Programs.* PhD thesis, Yale University, 1985.

[36] N. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, 1986.

[37] N. D. Jones and H. Søndergaard. A semantics-based framework for the abstract interpretation of Prolog. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 6, pages 123–142. Ellis Horwood, 1987.

[38] T. Kanamori. Abstract interpretation based on Alexander templates. TR 549, ICOT, 1990.

[39] T. Kanamori and K. Horiuchi. Type inference in Prolog and its application. TR 95, ICOT, 1984.

[40] T. Kanamori and T. Kawamura. Analyzing success patterns of logic programs by abstract hybrid interpretation. TR 279, ICOT, 1987.

[41] T. Kanamori, T. Kawamura, and M. Maeji. Logic program analysis by abstract hybrid interpretation. TR 485, ICOT, 1989.

[42] R. Kemp and G. Ringwood. Algebraic framework for the abstract interpretation of logic programs. In *Proceedings of the North American Conference on Logic Programming*, Austin, Texas, 1990. MIT Press.

[43] M. Kirschenbaum, A. Lakhotia, and L. Sterling. Skeletons and techniques for Prolog programming. Technical Report 89-170, Case Western Reserve University, 1989.

[44] J. Lever. Deriving argument properties through analysis of argument place dependency. Unpublished manuscript available from author, 1988.

[45] J. W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, 1987.

[46] C. K. Looi. *Automatic Program Analysis in a Prolog Intelligent Teaching System.* PhD thesis, University of Edinburgh, 1988.

[47] K. Marriott and H. Søndergaard. Bottom-up abstract interpretation of logic programs. In *Fifth Symposium and Conference on Logic Programming*, Seattle, Washington, 1988. MIT Press.

[48] K. Marriott and H. Søndergaard. Semantics-based dataflow analysis of logic programs. In G. Ritter, editor, *Information Processing 89.* North Holland, 1989.

[49] C. Mellish. The automatic generation of mode declarations for Prolog programs. Research Report 163, Department of Artificial Intelligence, University of Edinburgh, 1983.

[50] C. Mellish. Some global optimizations for a Prolog compiler. *Journal of Logic Programming*, 2(1):43–66, 1985.

[51] C. Mellish. Abstract interpretation of Prolog programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 8, pages 181–198. Ellis Horwood, 1987.

[52] C. Mellish. Using specialization to reconstruct two mode inference systems. Unpublished manuscript available from author, 1990.

[53] D. Michie. "Memo" functions and machine learning. *Nature*, 218:19–22, Apr 1968.

[54] P. Mishra. Towards a theory of types in Prolog. In *Logic Programming Symposium*, pages 289–298, Atlantic City, New Jersey, 1984. IEEE.

[55] A. Mulkers, W. Winsborough, and M. Bruynooghe. Analysis of shared data structures for compile-time garbage collection in logic programs. In D. H. D. Warren and P. Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 747–762, Jerusalem, Israel, 1990. MIT Press.

[56] W. R. Murray. *Automatic Program Debugging for Intelligent Tutoring Systems.* Morgan Kaufmann, 1988.

[57] K. Muthukumar and M. Hermenegildo. Determination of variable dependence information through abstract interpretation. In E. Lusk and R. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 166–185, Cleveland, Ohio, 1989. MIT Press.

[58] A. Mycroft. *Abstract Interpretation and Optimizing Transformations.* PhD thesis, University of Edinburgh, 1981.

[59] A. Mycroft. Polymorphic type schemes and recursive definitions. In G. Goos and J. Hartmanis, editors, *Sixth International Symposium on Programming*, pages 217–228, 1984.

[60] A. Mycroft and R. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:296–307, 1984.

[61] R. O'Keefe. Finite fixed-point problems. In J-L. Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, pages 729–743, Melbourne, Australia, 1987. MIT Press.

[62] D. Plaisted. The occur-check problem in Prolog. In *Logic Programming Symposium*, pages 272–28, Atlantic City, New Jersey, 1984. IEEE.

[63] R. Ramakrishan. Magic templates: A spellbinding approach to logic programs. In *Fifth Symposium and Conference on Logic Programming*, pages 140–159, Seattle, Washington, 1988. MIT Press.

[64] A. Wærn. An implementation technique for the abstract interpretation of Prolog. In *Fifth Symposium and Conference on Logic Programming*, pages 700–710, Seattle, Washington, 1988. MIT Press.

[65] H. Seki. On the power of Alexander templates. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 150–159, 1989.

[66] E. Shapiro. Concurrent Prolog: A progress report. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers*, chapter 5, pages 157–187. MIT Press, 1987. Volume 1.

[67] E. M. Soloway and B. Woolf. Problems, plans and programs. *SIGSCE Bulletin*, (12):16–24, 1980.

[68] H. Søndergaard. An application of abstract interpretation of logic programs: Occur check reduction. In B. Robinet and R. Wilhelm, editors, *ESOP 86 European Symposium on Programming*, pages 327–338, Saarbrucken, Federal Republic of Germany, March 17-19 1986. Springer Verlag. LNCS 213.

[69] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[70] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In *Proceedings of the Second International Conference on Logic Programming*, Uppsala, Sweden, 1984.

[71] H. Tamaki and T. Sato. Old resolution with tabulation. In E. Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, pages 84–98, London, England, 1986.

[72] S. Taylor, S. Safra, and E. Shapiro. A parallel implementation of FCP. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers*, chapter 39, pages 575–604. MIT Press, 1987. Volume 2.

[73] J. A. Thom and J. Zobel. Nu-Prolog reference manual. Technical Report 86/10, Machine Intelligence Project, Dept. Computer Science, University of Melbourne, 1987.

[74] M. van Emden and R. Kowalski. The semantics of logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[75] M. W. van Someren. What's wrong? understanding beginners' problems with Prolog. *Instructional Science*, 19(4/5), 1990.

[76] R. Waters. KBEmacs: a step toward the programmer's apprentice. Technical Report 753, MIT Artificial Intelligence Laboratory, 1985.

[77] W. Winsborough. Path-dependent reachability analysis for multiple specialization. In E. Lusk and R. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 133–153, Cleveland, Ohio, 1989. MIT Press.

[78] W. Winsborough. A simple analysis of shared structures. In *ICLP'91 Pre-Conference Workshop on Semantics-Based Analysis of Logic Programs*, 1991.

[79] H. Xia and W. K. Giloi. A new application of abstract interpretation in Prolog programs: Data dependency analysis. In *IFIP WG 10.0 Workshop on Concepts and Characteristics of Declarative Systems*, 1988.

[80] J. Zobel. Derivation of polymorphic types for Prolog programs. In J-L. Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, pages 817–838, Melbourne, Australia, 1987. MIT Press.

Appendix A

Examples

Here we give the results of analysing some predicates.

Reverse List

The first example is the standard quick reverse predicate. For this example we give each stage of our analysis.

We use an implementation of reverse which caused APROPOS2 to produce an incorrect analysis:

```
qr([], [B|B]).
qr([Head|Tail], [B|Temp]) :-
   qr(Tail, [B,Head|Temp]).
```

The implementation is unusual in that it uses the list constructor to pair the two parts of the accumulator. The first stage is to convert it into the normal form (see Section 1.6):

```
qr(A, B, C) :- A=[], B=C.
qr(A, B, C) :- A=[D|E], F=[D|C],
   qr(E, B, F).
```

The code is then analyzed by the abstract interpreter with a general abstract query with substitution:

```
qr(A,B,C) subst({{A},{B},{C}},{})
```

The following gives the raw output of the abstract interpreter for detecting inclusion and sharing relations. These annotations were obtained on the abstract interpreter by collecting the least upper bound of the current abstract substitutions when execution passed through the various program points:

```
qr(D,E,F):-   : subst({{D},{E},{F}},{})
    D=[]   : subst({{D},{E},{F}},{})
    E=F    : subst({{D},{E,F}},{})
qr(G,H,I):-   : subst({{G},{H},{I},{J},{K},{L}},{})
    G=[J|K]   : subst({{G},{G,J},{G,K},{H},{I},{L}},{})
    L=[J|I]   : subst({{G},{G,J,L},{G,K},{H},{I,L},{L}},{})
    qr(K,H,L)   : subst({{G},{G,H,J,L},{G,H,K},
                         {G,K},{H},{H,I,L},{H,L}},{})
```

The next stage is to extract the inclusion relation from this raw form. We require only the relations caused by each subgoal, though all the previously inferred relations still hold. Note that every variable includes itself. These extra elements in the relations are not made explicit in the annotations:

```
qr(D,E,F):-   :
    D=[]   :
    E=F    : E>>F, F>>E
qr(G,H,I):-   :
    G=[J|K]   : G>>J, G>>K
    L=[J|I]   : L>>I, L>>J
    qr(K,H,L)   : H>>I, H>>J, H>>L
```

From these inclusion annotations it is possible to detect programming techniques. The following shows the output from the technique detector described in Chapter 3:

```
Clause: 1
E - F - [DP-[E>>F]]

F - E - [DP-[F>>E]]

Clause: 2
H - I - [AC-[H>>L,L>>I],DP-[H>>L,L>>I]]
```

Each clause in the entire program is considered in turn and the detected techniques listed for each pair of variables in the head of each clause. In the above an accumulator pair (`AC`) and a dependency chain (`DP`) were detected in the second clause of qr/3 between the variables H and I in the clause head (that is the second and third arguments) where L is an intermediate variable. Since accumulators are a subclass of dependency chains these techniques will always be detected together in this way.

The program currently detects only the dependency chain, accumulator pair and difference structure (`Diff`) techniques.

For this program APROPOS2 fails to match the accumulator and produces a series of 'incorrect clause' errors even though its dynamic analysis shows that the student's program produces correct results. Our analysis, on the other hand, successfully detects the accumulator pair.

Append

The example here is the standard append/3 predicate:

```
app(D,E,F):- D=[], E=F.
app(G,H,I):- G=[J|K], I=[J|L],
   app(K,H,L).
```

```
append([], L, L).
append([H|L1], L2, [H|L3]) :-
   append(L1, L2, L3).
```

Again we start with the output of the abstract interpreter:

```
app(D,E,F):-   : subst({{D},{E},{F}},{})
    D=[]   : subst({{D},{E},{F}},{})
    E=F   : subst({{D},{E,F}},{})
app(G,H,I):-   : subst({{G},{H},{I},{J},{K},{L}},{})
    G=[J|K]   : subst({{G},{G,J},{G,K},{H},{I},{L}},{})
    I=[J|L]   : subst({{G},{G,I,J},{G,K},{H},{I},{I,L}},{})
    app(K,H,L)   : subst({{G},{G,I,J},{G,I,K,L},{G,K},
                                        {H,I,L},{I},{I,L}},{})
```

The resulting inclusion annotations are as follows:a

```
app(D,E,F):-   :
    D=[]   :
    E=F   : E>>F, F>>E
app(G,H,I):-   :
    G=[J|K]   : G>>J, G>>K
    I=[J|L]   : I>>J, I>>L
    app(K,H,L)   : I>>H, L>>H
```

And the techniques detected are:

```
Clause: 1
E - F - [DP-[E>>F]]

F - E - [DP-[F>>E]]

Clause: 2
I - H - [Diff-[I>>L,L>>H],DP-[I>>L,L>>H]]
```

Our analysis shows that the second clause of append/3 contains a difference structure (`Diff`) though only a simple one where the variable L is left uninstantiated in the tail of the output list which has only one element.

Flatten List

In the following examples we give only the code annotated with inclusion relations and the output of the techniques detector.

The standard Prolog example for flattening a nested list can be written as either an accumulator pair or a difference structure. There is only a small syntactic difference between these two programs. The following shows that our analysis can correctly distinguish between each version. Firstly, we take the difference structure version:

```
flatten_dl(C, D) :-
    E=[], flatten_dl(C, D, E).

flatten_dl(F, G, H) :- F=[], G=H.
flatten_dl(I, J, K) :- J=[I|K],
    A\=[], A\=[_|_].
flatten_dl(L, M, N) :- L=[O|P],
    flatten_dl(O, M, Q),
    flatten_dl(P, Q, N).
```

```
flatten_dl(L, FL) :-
    flatten_dl(L, FL, []).

flatten_dl([], L, L).
flatten_dl(X, [X|L], L) :-
    X\=[], X\=[_|_].
flatten_dl([H|T], L, L0) :-
    flatten_dl(H, L, L1),
    flatten_dl(T, L1, L0).
```

The annotations detected are as follows:

```
flatten_dl(C,D):-   :
    E=[]   :
    flatten_dl(C,D,E)   : D>>E

flatten_dl(F,G,H):-   :
    F=[]   :
    G=H   : G>>H, H>>G
flatten_dl(I,J,K):-   :
    J=[I|K]   : J>>I, J>>K
    I\=[]
    I\=[_|_]
flatten_dl(L,M,N):-   :
    L=[O|P]   : L>>O, L>>P
    flatten_dl(O,M,Q)   : M>>Q
    flatten_dl(P,Q,N)   : M>>N, Q>>N
```

The techniques detected are:

```
Clause: 1
Clause: 2
G - H - [DP-[G>>H]]

H - G - [DP-[H>>G]]

Clause: 3
J - I - [DP-[J>>I]]

J - K - [DP-[J>>K]]

Clause: 4
M - N - [Diff-[M>>Q,Q>>N],DP-[M>>Q,Q>>N]]
```

A difference structure is detected in the second and third arguments of the recursive clause. Next, the accumulator version:

```
flatten_ac(C, D) :-
   E=[], flatten_ac(C, D, E).

flatten_ac(F, G, H) :- F=[], G=H.
flatten_ac(I, J, K) :- J=[I|K],
   A\=[], A\=[_|_].
flatten_ac(L, M, N) :- L=[O|P],
   flatten_ac(P,Q,N),
   flatten_ac(O,M,Q).
```

```
flatten_ac(L, FL) :-
   flatten_ac(L, FL, []).

flatten_ac([], L, L).
flatten_ac(X, [X|L], L) :-
   X\=[], X\=[_|_].
flatten_ac([H|T], L, L0) :-
   flatten_ac(T, L1, L0),
   flatten_ac(H, L, L1).
```

The annotations detected are as follows:

```
flatten_ac(C,D):-  :
    E=[]  :
    flatten_ac(C,D,E)  : C-D>>E

flatten_ac(F,G,H):-  :
    F=[]  :
    G=H  : G>>H, H>>G
flatten_ac(I,J,K):-  :
    J=[I|K]  : J>>I, J>>K
    I\=[]
    I\=[_|_]
flatten_ac(L,M,N):-  :
    L=[O|P]  : L>>O, L>>P
    flatten_ac(P,Q,N)  : Q>>N
    flatten_ac(O,M,Q)  : M>>N, M>>Q
```

The techniques detected are:

```
Clause: 1
Clause: 2
G - H - [DP-[G>>H]]

H - G - [DP-[H>>G]]

Clause: 3
J - I - [DP-[J>>I]]

J - K - [DP-[J>>K]]

Clause: 4
M - N - [AC-[M>>Q,Q>>N],DP-[M>>Q,Q>>N]]
```

An accumulator pair is detected in the second and third arguments of the recursive clause.

Dutch Flag

The last example illustrates that the analysis can reliably detect even the more obscure instances of difference structures. The Dutch flag problem was first mention in Section 3.4.2. Here we give the results of analysing a simplified version where the input list contains only two types of tokens rather than three:

```
halfdutch(C,D) :- E=F,
    hdl(C,D,E,F).

hdl(G,H,I,J) :- G=[], H=I, J=[].
hdl(K,L,M,N) :-
    K=[O|P], O=white(Q), L=[O|R],
    hdl(P,R,M,N).
hdl(S,T,U,V) :-
    S=[W|X], W=blue(Y), V=[W|Z],
    hdl(X,T,U,Z).
```

```
halfdutch(Xs,WB) :-
    hdl(Xs, WB, B, B).

hdl([],WB,WB,[]).
hdl([w(X)|Xs],[w(X)|WB],W,B):-
    hdl(Xs,WB,W,B).
hdl([b(X)|Xs],WB,W,[b(X)|B]) :-
    hdl(Xs,WB,W,B).
```

The following shows the inclusion relationships detected:

```
halfdutch(C,D):-  :
    E=F  : E>>F, F>>E
    hdl(C,D,E,F)  : D>>E, D>>F

hdl(G,H,I,J):-  : I>>J
    G=[]  :
    H=I  : H>>I, H>>J, I>>H
    J=[]  :
hdl(K,L,M,N):-  : M>>N
    K=[O|P]  : K>>O, K>>P
    O=white(Q)  : K>>Q, O>>Q
    L=[O|R]  : L>>O, L>>Q, L>>R
    hdl(P,R,M,N)  : L>>M, L>>N, R>>M, R>>N
hdl(S,T,U,V):-  : U>>V
    S=[W|X]  : S>>W, S>>X
    W=blue(Y)  : S>>Y, W>>Y
    V=[W|Z]  : U>>W, U>>Y, U>>Z, V>>W, V>>Y, V>>Z
    hdl(X,T,U,Z)  : T>>U, T>>V, T>>W, T>>Y, T>>Z
```

The analysis correctly identifies the difference structure in the second and third arguments in the second clause of hdl/4:

```
Clause: 1
Clause: 2
H - I - [DP-[H>>I]]

H - J - [DP-[H>>J]]

I - H - [DP-[I>>H]]

Clause: 3
L - M - [Diff-[L>>R,R>>M],DP-[L>>R,R>>M]]

L - N - [Diff-[L>>R,R>>N],DP-[L>>R,R>>N]]

Clause: 4
T - U - [DP-[T>>U]]

T - V - [DP-[T>>V]]
```